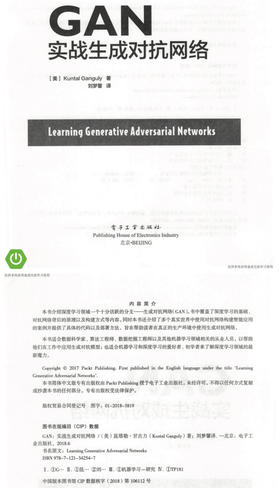
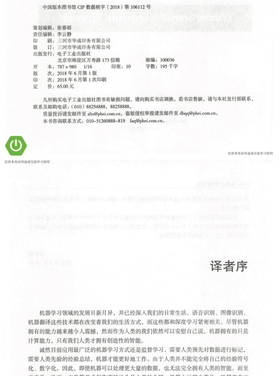




译者序

机器学习领域的发展日新月异，并已经深入我们的日常生活，语音识别、图像识别、机器翻译这些技术都在改变着我们的生活方式。而这些都和深度学习紧密相关。尽管机器都有能力越来越令人震撼，然而作为人类的我们依然可以安慰自己说，机器拥有的只是计算能力，只有我们人类才拥有创造性的智能。

诚然目前应用最广泛的机器学习方式还是监督学习，需要人类预先对数据进行标记，需要人类先验的经验总结，机器才能更好地工作。而人类并不能完全将自己的经验尽可化、数字化，因此，即使机器可以处理更大量的数据，也无法完全拥有人类的智能。而至于语义、情感、文学等艺术创作更被认为是人类的专属，机器永远做不出来。机器离真正的智能还有很大的距离。

而生成对抗网络（GAN）的出现，让事情发生了变化。生成对抗网络采用无监督学习的方式，在训练数据中进行学习，在不需要人工对数据标签行标记的情况下就可以产生令人惊叹的效果。从维基百科的词条学习专业知识，然后写出一个令普通人可以理解的科普文章，模仿知名画家的作品并学习他们的风格进行再创作，这种看似只有人类这样拥有智能和创造性的物种才会做的事情，机器学习现在真的可以做到了！

阅读本书的作者 Kuntal Ganguly 就将在本书中向你介绍这个神奇的技术，生成对抗网络也是如今深度学习里最为前沿也是最为令人着迷的领域之一。本书从生成模型的基础讲起，逐步介绍了各种生成对抗网络的变体和使用场景，除了深度学习相关的理论，书中还有大量基于现实世界的使用案例和代码，给读者提供了充足的实践方法，相信通过

本书你可以快速地领略到生成对抗网络的魅力及其无限的可能性，并将其应用到你的实际工作之中。

非常荣幸获得电子工业出版社张春雨先生的邀请来负责本书的中文翻译，在翻译的过程中我深感自己的中英文能力十分有限，加之自己在机器学习领域也是个新手，经译文仍有不足之处，十分期待大家的指正。

刘梦馨
2018 年 5 月 19 日

作者简介

Kuntal Ganguly 是一位大数据分析工程师，其利用机器学习和大数据框架来搭建大规模数据驱动系统。Kuntal 具有 7 年的大数据和机器学习应用构建经验。

Kuntal 为云上客户提供搭建实时分析系统的解决方案。这其中利用到了托管式的云服务和开源 Hadoop 生态系统工具，诸如 Spark、Kafka、Storm、Solr 以及机器学习和深度学习框架。

Kuntal 也喜欢亲自动手参与软件的开发过程，并且曾经独自一人完成了多个大规模分布式应用从构思、架构、开发一直到部署的整个过程。他也是一位机器学习和深度学习的从业者，十分热衷于构建智能应用。

下面是他的 LinkedIn 主页链接：`https://www.linkedin.com/in/kuntal-ganguly-59564088/`。

十分感谢我的母亲 Chitra Ganguly 以及我的父亲 Gopal Ganguly 对我的爱和支持，以及他们多次教导我要努力工作。从他们身上学到的点点滴滴对我的整个人生都有巨大的帮助。同时，我也想感谢这么多年来陪伴我的朋友、同事以及导师。

审稿人介绍

Max Strakhov 是一位具有 8 年计算机编程工作经验以及 4 年机器学习工作经验的软件工程师和研究者。他曾在 Google 和 Yandex 工作，目前是 AURA Devices LLC 的联合创始人及 CTO。

他对深度学习尤其是生成网络在人工智能领域的应用有着深厚的兴趣。他在自己的博客 `http://monnoroch.github.io/`中记录了关于深度学习、软件工程以及其他相关技术的分享。

我相信数据科学和人工智能将会赋予我们超能力。

目录

前言

本书中提到的概念和模型可以帮助你快速地建立深度网络，创造性地利用无监督学习方法来完成监督学习领域的任务，例如图像分类。

利用生成网络的基本概念，你会学到如何从无标签的数据中生成逼真的图像，根据文本的描述信息制作图像，以及探索跨域的风格迁移以及域之间的关系。

本书的内容

第 1 章　深度学习概述，以一种简单的没有太多数学和公式的方式介绍了深度学习中最新的常用概念和术语。同时也展示了深度学习这些年来的演化，以及深度学习如何利用生成模型逐渐进入无监督学习领域的过程。

第 2 章　无监督学习 GAN，介绍生成对抗网络的工作原理以及如何构建生成对抗网络。同时还会介绍深度学习网络如何应用于半监督学习领域，以及如何应用它们来进行图像的生成和创作。GAN 的训练十分困难，在本章中会介绍一些提升训练/学习效果的技术。

第 3 章　图像风格跨域转换，介绍如何利用简单但强大的 CGAN 和 CycleGAN 模型进行创意制作。本章展示了利用条件 GAN 根据某种特定风格或者条件来制作图像。本章还讨论了如何利用 BEGAN 进行网络固化来避免模型坍塌的问题。最后，展示了如何利用 CycleGAN 来进行跨域转换（苹果变成橘子，马变成斑马）。

第 4 章　从文本构建逼真的图像，展示了利用最新的层叠 GAN 技术将从文本生成图像的问题拆分成两个可控的子问题，并利用 StackGAN 进行处理。本章还会展示如何利用

DiscoGAN 成功地进行图像风格的跨域转换，包括以边框的图像为输入来输出手提包的图像，以及对名人的图像进行性别转换。

第 5 章　利用多种生成模型生成图像，介绍预训练模型的概念并讨论了利用大规模分布式系统 Apache Spark 运行深度学习和生成模型。接下来我们会利用 GAN 和预训练模型来提升低品质图像的清晰度。最后，我们会介绍用 DeepDream 和 VAE 等其他生成模型来生成图像和风格转换。

第 6 章　将机器学习带入生产环境，介绍多种将以机器学习和深度学习为基础的智能应用部署到生产环境的方法，包括基于数据中心的部署方式，以及基于云环境的容器化微服务部署方式，或基于无服务器技术的部署方式。

阅读本书的准备工作

本书中所用到的工具、库以及数据集都是开源的并可免费获取。本书中提到的云环境会提供一些免费试用来给用户进行评测。通过本书以及其他一些机器学习和深度学习的资料，读者将会发现生成对抗网络所拥有的无限创造力。

你需要安装 Python 并用 `pip` 安装一些相关的 Python 软件包来有效地运行本书中提供的样例代码。

本书面向的读者

本书所面向的读者是那些希望了解生成模型的威力以及从无标记的源数据和噪声中制作逼真图像的机器学习专家以及数据科学家。

拥有足够机器学习和神经网络知识并且能熟练使用Python进行编程的读者可以通过本书学会如何从文本生成图像，自动发现相似域之间的关联，并能基于深度学习和生成模型来探索无监督学习领域的问题。

约定惯例

在本书中，你会发现我们将用不同风格的数字和文本来区分不同的信息。下面是一些文本风格的样例，以及对应含义的解释。

导入相关软件模块的代码块如下所示：

```
<div class="packt_code">
nsamples=6
Z_sample = sample_Z(nsamples, noise_dim)
y_sample = np.zeros(shape=[nsamples, num_labels])
y_sample[:, 7] = 1 # generating image based on label
samples = sess.run(G_sample, feed_dict={Z: Z_sample, Y:y_sample})
</div>
```

新出现的术语和**重要词汇**会被加粗显示。如果你使用屏幕阅读，屏幕中在菜单和对话框中出现的词汇，在文本中以下面的形式展现，例如：“单击**下一页**按钮将进入下一页”。

警告和重要的笔记出现在这样的方框中。

提示和技巧出现在这样的方框中。

读者服务

轻松注册成为博文视点社区用户（www.broadview.com.cn），扫码直达本书页面。

- **下载资源**：本书提供示例代码及资源文件，可在下载资源处下载。
- **提交勘误**：您对书中内容的修改意见可在提交勘误处提交，若被采纳，将获赠博文视点社区积分（在您购买电子书时，积分可用来抵扣相应金额）。
- **交流互动**：在页面下方读者评论处留下您的疑问或观点，与我们和其他读者一同学习交流。

页面入口：http://www.broadview.com.cn/34254

1 深度学习概述

深度神经网络目前在解决多种问题，在图像识别、语音识别、机器翻译和自然语言处理等多个领域可以提供效果和人工类似的解决方案。

在本章中，我们将会回顾神经网络这种从生物学获得启发的架构这些年来是如何不断演化的。作为之后几章的基础，接下来我们将会进入深度学习相关的一些最重要的概念和术语。最后我们将通过生成模型来了解深度网络创造性特性背后的“原力”。

本章将会包含以下内容：

- 深度学习的演化。
- 随机梯度下降、修正线性单元（ReLU）、学习速率和一些其他概念。
- 条件网络、递归神经网络以及 LSTM。
- 判别模型和生成模型之间的区别。

1.1 深度学习的演化

和神经网络相关的大量工作诞生于 20 世纪八九十年代，然而当时的计算机运行得十分缓慢，数据量也很小，科研人员并没有发现在现实世界中能够使用神经网络的场景。因此，在 21 世纪的头 10 年，神经网络几乎从机器学习的世界中消失了。直到最近几年，最先在 2009 年的语言识别领域，接下来在 2012 年的计算机视觉中，神经网络凭借优异的表现重新归来（同时还带来了 LeNet、AlexNet 等）。这期间什么发生了变化？

大量的数据（Big Data）以及廉价高速的 GPU 改变了这一切。现如今神经网络已经进入了世界的每一个角落。如果你正在做任何和数据相关的工作，则不管是分析还是预测，

深度学习都将会是你必须要了解的技术。

下面看一下图 1-1。

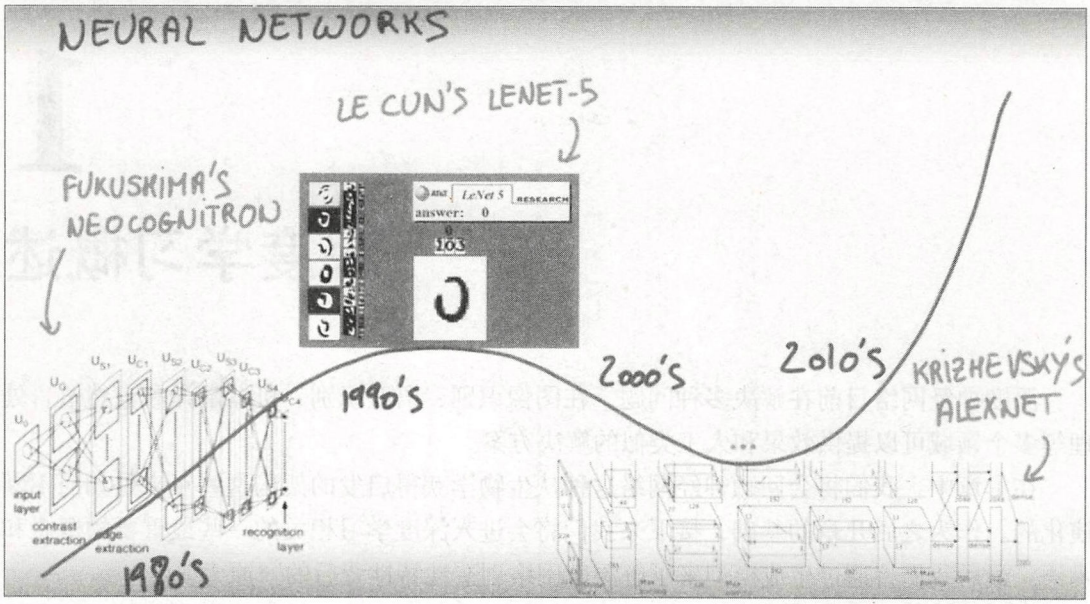


图1-1：深度学习的演化

深度学习是机器学习中十分令人振奋的一个分支，它利用大量的数据来训练计算机去做一些之前只有人类才能做的事情，诸如如何分辨图像中有哪些物体，分辨人们在打电话时对话的内容，将文档翻译成另一种语言，帮助机器人探索世界并对各种事情及时响应，等等。深度学习成为解决机器认知问题最为核心的工具，并且是计算机视觉和语言识别领域当下最为优秀的技术。

现如今大量的工作都将深度学习作为机器学习工具链中最为核心的一部分。由于深度学习在大规模数据量和解决复杂问题中能发挥最大的优势，因此，Facebook、Baidu、Amazon、Microsoft 和 Google 都在产品中使用到了深度学习。

深度学习通常是“深度神经网络”的代称，深度神经网络中包含了多个层，每一层又是由多个节点组成的。计算发生在每个节点，每个节点会将输入以及一系列的参数和权重进行计算，将输入进行增强或者抑制。这些输入-权重组合的结果将会被汇总，它们的总和会被传入一个激发（activation）函数，通过这个函数来确认这个汇总值该如何在整个网络中进行传递，以及会对最终的结果产生怎样的影响。例如在一个分类任务中，这个值会对分类结果产生何种影响。神经网络的每一层包含一排的节点，通过输入数据的不同来决

定每个节点是开启的还是关闭的，并将输入在网络中进行传递。初始输入经过第 1 层的处理变为第 2 层的输入，依此类推。图 1-2 是一个神经网络的示意图。

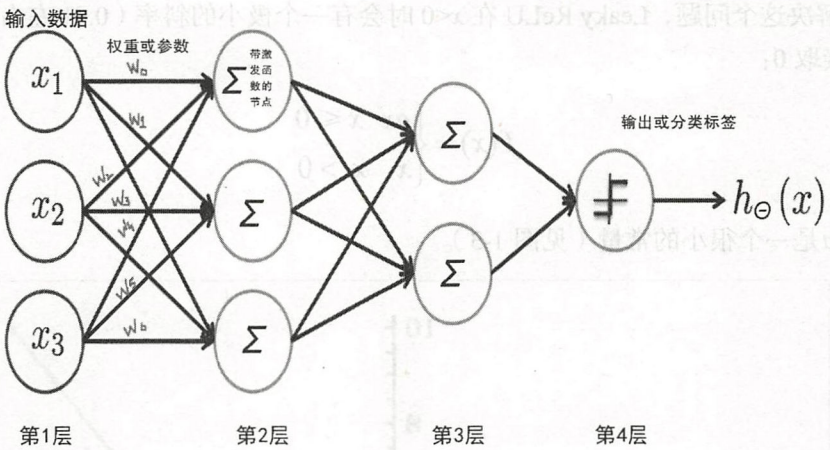


图1-2：神经网络示意图

下面让我们来了解一下深度神经网络里的一些概念和术语。

1.1.1 sigmoid 激发

sigmoid 激发函数在神经网络中产生范围在(0,1)之间的输出结果，其中参数偏移量 a 表示当输入为 a 时 sigmoid 会生成 0。

当输入数据 x 在特定范围内时，sigmoid 函数在梯度下降过程中的表现十分优异。但是对于 x 很大而 y 是常量的情况，由于 dy/dx（梯度）始终是 0，因此会导致**梯度消失**（vanishing gradient）的问题。

由于梯度是 0，因此将梯度和损失（实际值–预测值）相乘依旧是 0，这最终会导致神经网络停止学习。

1.1.2 修正线性单元（ReLU）

一个神经网络可以通过线性分类器和非线性函数组合而成。修正线性单元也被称作**线性整流单元**（Rectified Linear Unit，ReLU），其近些年来成了一种热门技术。它的计算公式为 $f(x)=\max(0,x)$。换而言之，该函数只取大于 0 的输入数据。不好的一面是，ReLU 单元

这种更新权重的方式导致它在训练过程中十分脆弱，一些神经元可能在任何数据上都不会再被激发，从这个单元点经过的梯度，从某个数据点之后就会一直是 0。

为了解决这个问题，Leaky ReLU 在 $x<0$ 时会有一个极小的斜率（0.01 左右）而不是当 $x<0$ 时直接取 0：

$$f(x)=\begin{cases} ax & x\leqslant 0 \\ x & x>0 \end{cases}$$

其中 a是一个很小的常量（见图 1-3）。

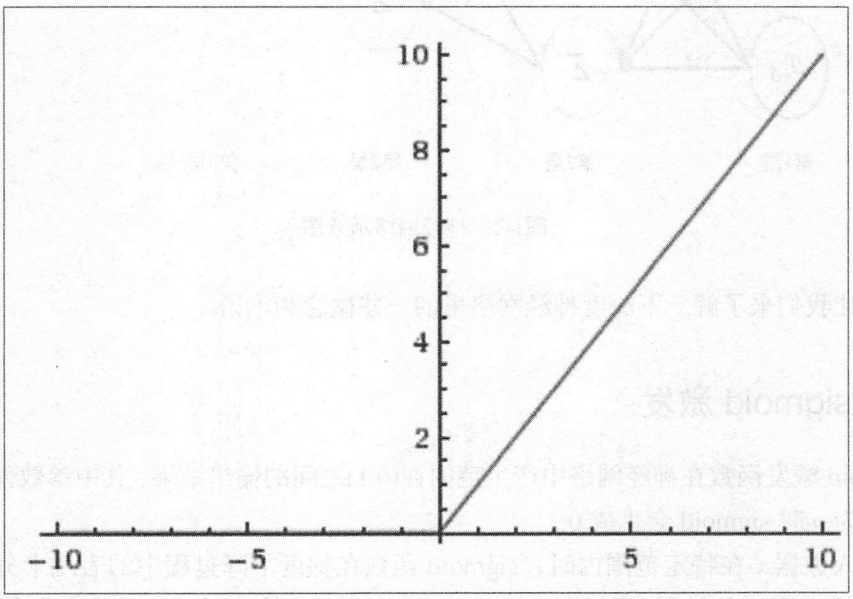


图1-3：修正线性单元

1.1.3 指数线性单元（ELU）

ReLU 激发的平均值不是 0，这会导致网络学习过程中遇到困难。**指数线性单元**（Exponential Linear Unit，ELU）在 x 为正数时的表现和 ReLU 相同，当 x 为负数且 a=1（超参数 a 控制输入为负数时的 ELU 取值范围）时将结果的下界限制为−1。这个特性使得激发函数的平均值趋向于 0，这可以使得网络更加健壮，能够更好地应对噪声。

1.1.4 随机梯度下降（SGD）

大批量地进行梯度下降训练是一件十分复杂的事情。因为当你的数据集十分大时，计算量也会特别大，同时如果你希望计算损失结果保留 n 位精度的浮点数，那么在梯度计算的时候就需要花 3 倍的时间。

但是在实践中，由于我们经常能从更多的训练数据中获得更好的训练结果，因此希望训练大量的数据。由于梯度下降需要迭代执行很多次，这就意味着在每一次迭代中即使只是更新一个参数，我们也需要在所有数据集上进行计算，然后在所有数据集上将这个过程迭代几十或上百次。

除了每次迭代在这个数据集上计算损失，我们还可以只在整个数据集中随机选取一部分来计算平均损失，例如每次迭代只取 1 ~ 1000 个数据进行损失计算。这种计数被称为**随机梯度下降**（Stochastic Gradient Descent，SGD）。由于 SGD 在数据量和模型规模上都有着很好的扩展性，因此其已经被称为深度学习中的核心技术。

SGD 有大量的超参数需要精心调优，例如初始化参数、学习速率参数、衰变参数以及动量参数，也因此 SGD 被称为一种黑魔法。

AdaGrad 对 SGD 进行了简化修改，可以隐式地对动量、学习速率和衰变进行调节。通过 AdaGrad 可以使得学习过程对超参数的调节不是那么敏感，但是它的训练效果通常要比精心调优过的 SGD 效果差一些。如果你只是想让模型可以工作，那么 AdaGrad 依然是一个很好的选择（见图 1-4 和图 1-5）。

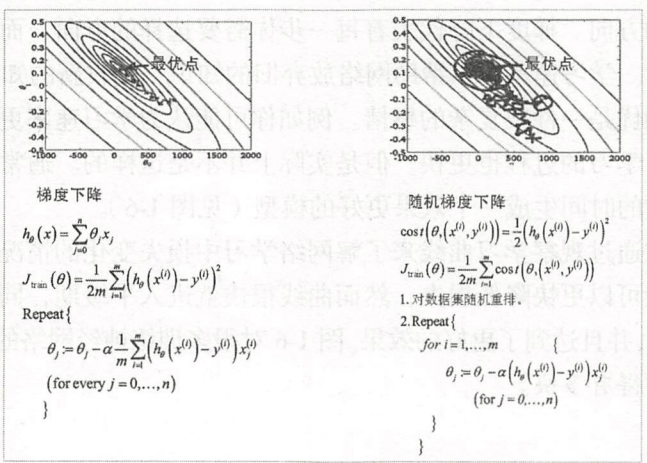


图1-4：批量梯度下降以及SGD的损失计算过程

（来源：https://www.coursera.org/learn/machine-learning/lecture/DoRHJ/stochastic-gradient-descent）

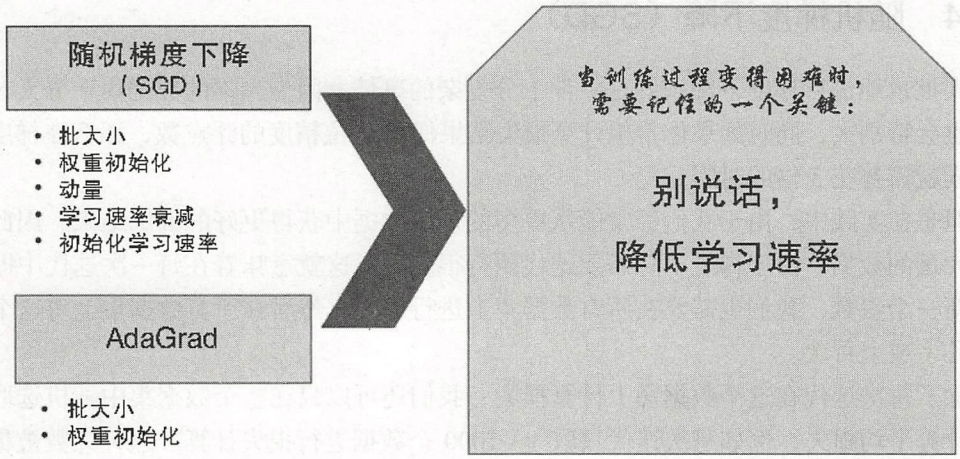


图1-5：随机梯度下降和AdaGrad

在图 1-4 中你会发现，批量梯度下降的方法很快找到了最优点；而对于 SGD 的计算过程，由于在迭代过程中每次随机选取一部分数据，因此会在最优点附近来回振荡。然而在实践中 SGD 通常会表现得更好，而且收敛得更快。

1.1.5 学习速率调优

深度学习的损失（loss）函数可以被比喻成一个平面，在其中网络的权重代表着每一步移动可以选择的方向。梯度下降代表着每一步你需要选择的方向，而学习速率代表着你每一步需要走多远。学习速率可以帮助网络放弃旧的知识，接收新的知识。

学习速率的调优是一件很复杂的事情。例如你可能认为学习速率更高意味着模型可以学到更多的知识，学习的过程也更快。但是实际上并不是这样的。通常你会发现降低学习速率反而能以更短的时间生成一个效果更好的模型（见图 1-6）。

你可能会尝试通过观察学习曲线来了解网络学习中损失变化的情况。在这里更高的学习速率在起始阶段可以更快降低损失，然而曲线很快就进入平缓期，同时在更低的学习速率下损失持续降低，并且达到了更好的效果。图 1-6 对很多训练神经网络的人来说都很熟悉，千万不要相信你学得有多快。

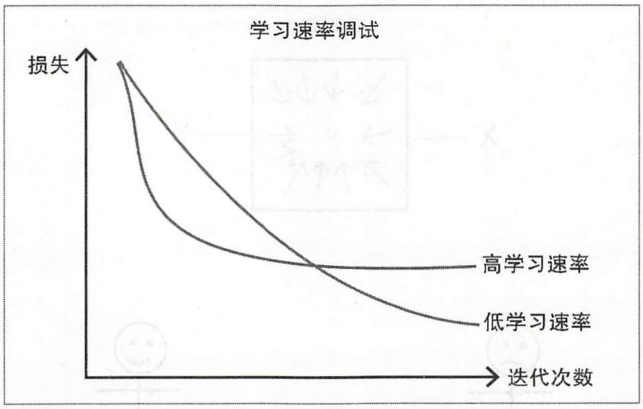


图1-6：学习速率

1.1.6 正则化

避免过拟合的最佳方法就是观察模型在验证集的性能指标，如果性能指标不再提升就停止训练。这种方法也被称为提前结束，这也是避免神经网络在训练集上过度优化的一种方法。另一种方法就是应用正则化，正则化意味着需要在网络中加入一些人工的约束条件来隐式地降低自由参数的数量，同时使得优化过程不会变得太困难（见图 1-7）。

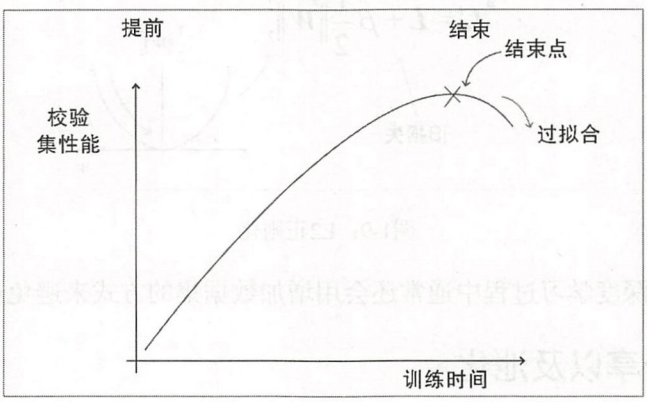


图1-7：提前结束

我们利用图 1-8 中弹力裤的比喻来描述这个问题。弹力裤可以很好地适配当前的情况，而且由于它具有弹性，因此对其他的情况也有很好的适应性。深度学习中的“弹力裤”被称为 **L2 正则化**。它的思想是通过给损失加入另外一个参数来惩罚那些数值大的权重（见图 1-9）。

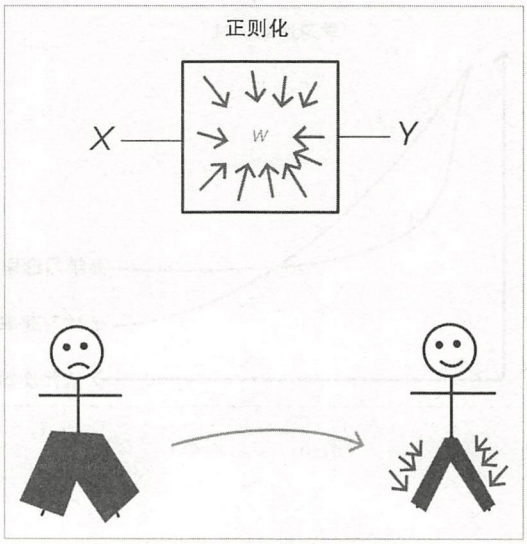


图1-8：用弹力裤来比喻正则化

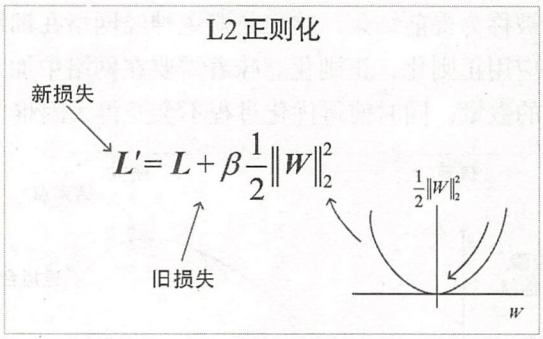


图1-9：L2正则化

在现实世界的深度学习过程中通常还会用增加数据集的方式来避免过拟合。

1.1.7 权重分享以及池化

如果图像里面有一只猫，不管这只猫在图像的哪个位置，它都依然是一张有一只猫的图像。如果神经网络分别学习位于左下角和位于右下角的猫，那么其将会花费大量的工作才能学会判断图像里是否有一只猫。但是不管猫在图像的左边还是右边，图像和图像里的物体绝大部分是相同的，我们称之为**平移不变性**（translation invariance，见图 1–10）。

神经网络中解决这个问题的方法被称为**权重分享**（weight sharing，见图 1–11）。当网

络得知两个输入包含相同的信息时，就可以将两个输入进行联合训练并分享其权重。权重分享是一个十分重要的思想。统计上的不变性不随着时间或空间的变化而改变，其存在于各个角落。对于图像来说，权重分享的思想将引导我们进入卷积网络。对于常规的文本序列，权重分享的思想将引导我们进入递归神经网络。

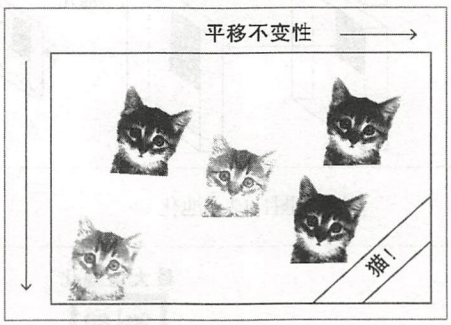


图1-10：平移不变性

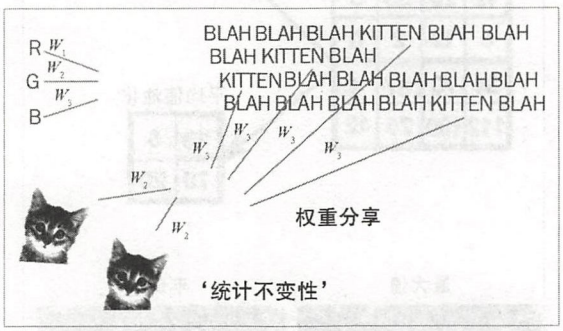


图1-11：权重分享

为了减少在卷积金字塔中的特征空间维度，可以进行一小步的尝试，然后获取所有邻近的卷积并将它们进行组合，这种方式也被称为**池化（pooling，见图 1–12）**。

在图 1-13 中所展示的最大值池化中，对于特征图中的每一个点，计算所有和它邻近点中数值最大的一个作为最终结果。最大值池化有着自己的优点。首先它没有增加参数个数，所以你不必担心会增加过拟合的风险。其次它在现实表现中通常都很优秀，会产生很准确的模型。但是由于卷积以较小的步长运行，因此训练的计算代价会很高。最大值池化通常可以提取出最重要的特征，而平均值池化由于考虑了所有的特征并进行了平均化，其会包含与任务无关的信息，因此通常在物体识别方面表现得不如最大值池化。

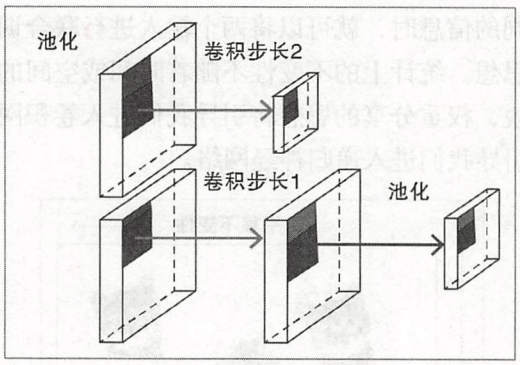


图1-12：池化

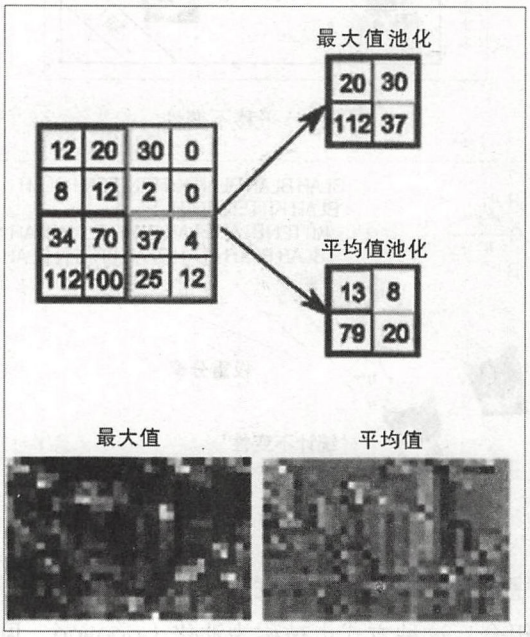


图1-13：最大值池化和平均值池化

1.1.8 局部感受野

一个对局部数据结构进行编码的简单方法是，连接多个邻近输入神经元的子矩阵，生成一个新的属于下一层的隐式神经元，这个新的隐式神经元代表了一个局部感受野。以CIFAR-10图像为例，每个图像的输入特征空间为[32×32×3]。如果感受野（过滤窗口）为4×4，那么在卷积层的每个神经元就会有一个对应输入空间[4×4×3]大小的区域权重，总

共有 4×4×3=48 个权重值（以及一个偏移量参数）。由于输入特征的深度（或者通道的数量：RGB）是 3，因此连接的深度空间也必须是 3。

1.1.9 卷积网络（ConvNet）

卷积网络（Convolutional Network，ConvNet）是一种在空间上分享参数或者权重值的神经网络。一个图像可以由一个扁平的薄饼来表示，具有高度、宽度和深度或者通道的数量（对于 RGB，有红绿蓝 3 个通道，深度是 3；对于灰度图，深度是 1）。

接下来我们进入一个小型的神经网络，这个网络权重不会变化，其通过输入图像产生 K 个输出（见图 1-14 和图 1-15）。

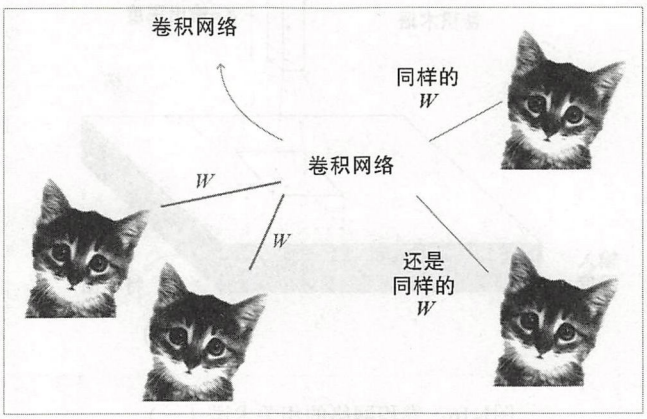


图1-14：空间上的权重分享

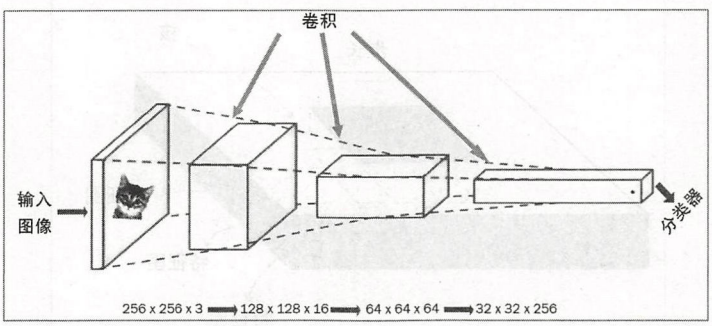


图1-15：多层卷积的卷积金字塔

在输出端，一个不同高度、宽度和深度的图像会被生成，这个操作被称为卷积。

一个卷积网络通常是一个深度网络，每一层对前一层进行卷积生成一种金字塔状的网

络结构。可以从前面的图 1-15 中看到，网络以一张 3 个维度（高度 × 宽度 × 深度）的图像作为输入，逐步地进行卷积来缩减图像平面空间上的维度，同时增加深度，这也是卷积这个词字面上所表达的含义。下面再让我们来了解一些卷积网络里的常用术语。

图像深度的每一层被称为一个特征，卷积核用来将 3 个特征图映射到 K 个特征图。步长是每一次需要过滤的像素点的个数。根据不同步长的区块，生成的结果大小也会不同。步长为 1，则输出大小和输入大小基本一致；步长为 2，输出大小大约是输入的 2 倍。对于一个有效的滑动区块，过滤器的范围不会超出图像的边界。而对于固定区块，过滤器会超出边界的范围，并且填充 0 值，使得输出特征图和输入特征图大小一致（见图 1-16 和图 1-17）。

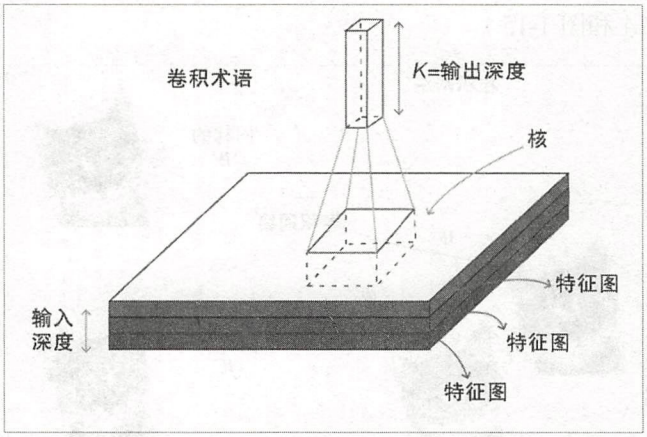


图1-16：卷积网络的相关术语（一）

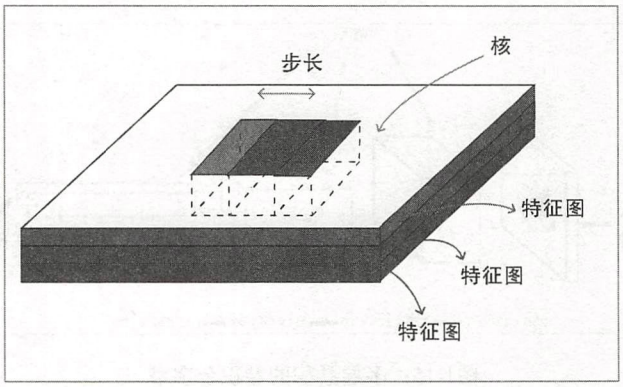


图1-17：卷积网络的相关术语（二）

1.2 逆卷积/转置卷积

对于计算机视觉的应用，如果需要最终生成图像的清晰度高于输入图像的清晰度，则逆卷积/转置卷积是这个领域事实上的标准。该卷积层在 GAN、超分辨率图像、图像深度估算、光流估算等应用场景中有着大量的应用。

CNN 通常采用下采样，也就意味着其会产生清晰度低于输入的结果；而逆卷积采用上采样，可以获得与输入相同清晰度的输出。需要注意，由于最基本的上采样会丢失一些细节，因此更好的选择是使用预先训练的上采样卷积层，并在训练过程中修改对应的参数。

TensorFlow 中的方法如下：

```
tf.nn.conv2d_transpose(value,filter,output_shape,strides,padding,name)
```

1.2.1 递归神经网络和 LSTM

递归神经网络（Recurrent Neural Network，RNN）的核心思想是在时间尺度上共享参数值。想象一下你拥有一个事件的序列，在某一个时间点你想确定接下来会发生什么。如果这个序列是相对固定的，你就可以在每个时间点上使用相同的分类器，这可以使任务大幅简化。但是由于它是一个序列，因此你也希望将这个时间点之前发生的事件考虑进来。

RNN 是一个汇总过去的结果并给当前的分类器提供信息的模型。它最终会生成一个具有相对简单重复模式的网络，每一个分类器除了和当前的输入进行连接外，还会连接过去每一步的结果，这也被称为递归连接，如图 1-18 和图 1-19 所示。

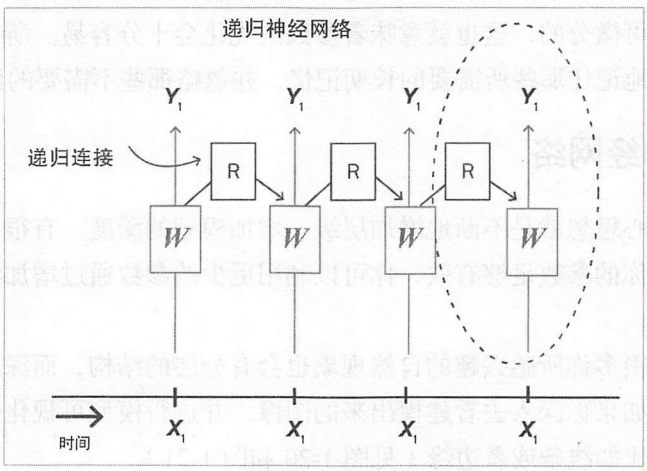


图1-18：递归神经网络

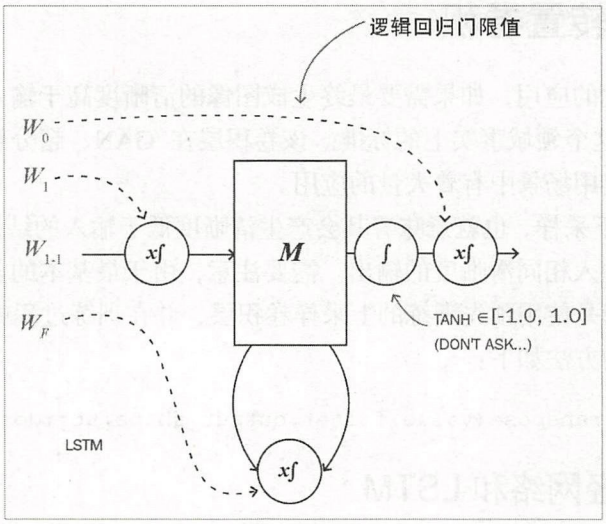


图1-19：长短期记忆（LSTM）网络

LSTM 是**长短期记忆**（long short-term memory）的简写。从概念上讲，一个递归神经网络包含一系列重复的单元，每个单元接收一个过去的输入、一个当前的输入，产生一个新的预测，并将结果连接到未来。在这之间的是简单的由权重和线性函数组成的一层层网络。

在图 1-19 所示的 LSTM 中，每一层的门限值是由一个小型的逻辑回归器控制的。每个逻辑回归器拥有一系列共享的参数并且通过一个额外的双曲线函数将输出值分布到–1 和 1 之间。同时它也是可微分的，这也就意味着参数的优化会十分容易。所有的这些小的门限可以帮助模型更好地记住那些所需要的长期记忆，并忽略那些不需要的记忆。

1.2.2 深度神经网络

深度学习的核心思想就是不断地增加层级、增加模型的深度。有很多理由去这样做。一个理由是，如果你的参数足够有效，你可以利用更少的参数通过增加深度而不是广度来获得性能上的提升。

另一个理由是很多你所感兴趣的自然现象也会有分层的结构，而深度模型的特性可以很好地与其匹配。如果你深入去看建模出来的图像，并进行模型可视化，就会在底层发现一些常见的元素，比如线条或者边缘（见图 1-20 和图 1-21）。

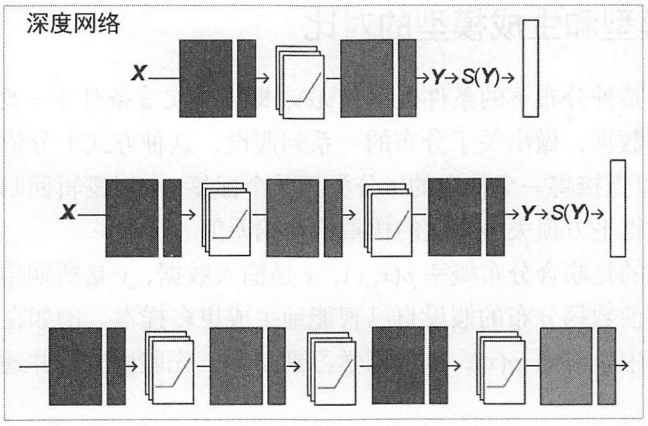


图1-20：深度神经网络

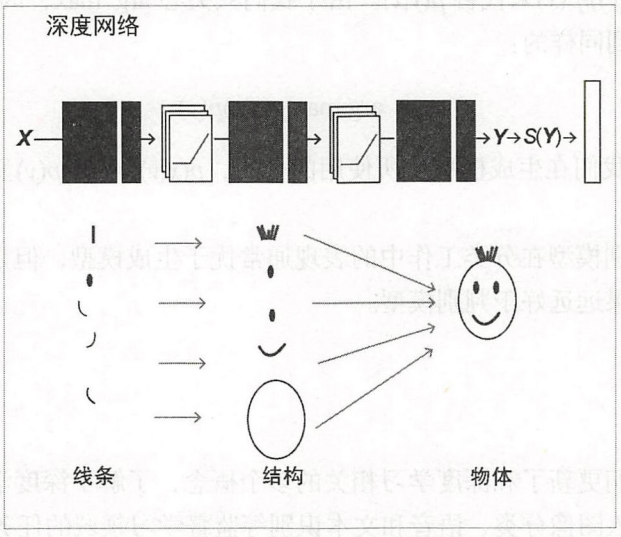


图1-21：捕获到图像结构的网络层

ConvNet 的一个常见架构是用一些和顶层数据全连接的层代替使用最大值池化的卷积层。第一个这种架构模型是著名的 LeNet-5，它由 Yann Lecun 在 1998 年进行字符识别时所使用。

现代的卷积网络，例如因在 2012 年赢得物体识别竞赛 ImageNet 而闻名的 AlexNet，使用了类似的架构，只使用了很少的卷积层。另一个著名的池化方法是平均值池化。其通过对特定位置窗口范围内的像素取平均值而不是最大值来做池化。

1.2.3 判别模型和生成模型的对比

判别模型学习某种分布下的条件概率 $p(y|x)$，即在特定 x 条件下 y 发生的概率。一个判别分类器通过观察数据，做出关于分布的一系列假设，这种方式十分依赖于数据的质量。概率分布 $p(y|x)$可以直接将一个特定的 x 分类到某个标签 y。以逻辑回归为例，我们所需要做的只是学习出使得平方损失最小化的权重值和偏差值。

生成模型学习的是联合分布概率 $p(x, y)$，x 是输入数据，y 是所期望的分类。一个生成模型可以根据对当前数据分布的假设自己智能地生成更多样本。例如在朴素贝叶斯模型中我们既可以从数据中学习到 $p(x)$，也可以学习到 $p(y)$、先验概率，并通过数据可以学习到最优可能的 $p(x|y)$。

一旦我们拥有了 $p(x)$、$p(y)$和 $p(x|y)$，那么 $p(x,y)$也可以轻松得到。根据贝叶斯规则，我们可以利用$(p(x|y)p(y))/p(x)$来代替 $p(y|x)$。由于我们只关心 arg max，因此分母可以被省略，我们会对每个 y 得到同样的：

$$\arg\max p(x|y)p(y)$$

这个公式就是我们在生成模型中所使用的公式，$p(x,y)=p(x|y)p(y)$显式地对每一类情况的分布进行了建模。

在现实中，判别模型在分类工作中的表现通常优于生成模型，但是在一些创造性的工作中生成模型的效果远远好于判别模型。

1.3 总结

到目前为止我们更新了和深度学习相关的多个概念，了解了深度学习如何通过极富创造力的生成模型进入图像分类、语音和文本识别等监督学习领域的任务。在第 2 章中我们会看到深度学习如何通过**生成对抗网络**（Generative Adversarial Network，GAN）在无监督领域解决令人惊叹的创造性任务。

2

无监督学习 GAN

随着生成模型的不断发展，神经网络除了进行图像识别外，现在也可以用来生成音频和逼真的图像。

在本章中我们将会通过目前最为先进的算法**生成对抗网络**（Generative Adversarial Network，GAN）来探索深度学习创造力的本质。你将通过一些可以直接上手的例子来学习，这其中你将会利用深度网络的生成特性通过现实世界的数据集（MNIST 和 CIFAR）来生成逼真的图像。同时你也将会了解如何通过半监督学习来解决在深度网络无监督学习中遇到的问题。在本章的最后一部分，你将会了解一些训练过程中常常碰到的问题以及使用 GAN 模型的一些实用技巧。

本章将会包含以下内容：

- 什么是 GAN，它的应用、建议以及技巧。
- 通过 TensorFlow 上的两层神经网络生成图像来解释 GAN 相关的概念。
- 通过 Keras 运行**深度卷积 GAN**（DCGAN）生成图像。
- 通过 TensorFlow 实现半监督学习。

2.1 利用深度神经网络自动化人类任务

最近几年，深度神经网络在图像识别、语音识别以及自然语言理解方面的应用有了爆炸式的增长，并且都达到了极高的准确度。

目前最先进的深度神经网络算法可以通过数据学习到高度复杂的模型和模式，它们的能力令人印象深刻。然而人类可以做能力远超出图像识别和语音识别的任务，而这些任务想要通过机器进行自动化似乎还是天方夜谭。

让我们来看一下那些需要利用到人类创造力（至少现在还需要）的例子：

- 通过学习维基百科上的文章来训练一个人工智能作者，以一种通俗易懂的方式写一篇面向社区解释科学概念的文章。
- 创造一个可以通过学习著名画家的作品集合来模拟他的风格进行创作的人工智能画家。

你觉得机器现在可以完成这种类型的任务吗？答案可能会出乎你的意料，答案是“可以”。

毫无疑问，这些都是很困难的任务，但是 GAN 使得这些任务的解决变得可能。

深度学习的领军人物，Facebook AI 部门的主管 Yann LeCun 曾经说过：

> 生成对抗网络（GAN）及其变种已经成为最近 10 年以来机器学习领域最为重要的思想。

如果你对 GAN 这个名称心有畏惧，不要担心，等到本书结束之际，你将会精通利用 GAN 解决现实世界问题的技术。

2.1.1 GAN 的目的

一些生成模型可以从模型的分布中生成样本。GAN 也是生成模型的一种，主要用于通过分布生成样本。

你可能想知道为什么生成模型值得学习，尤其在了解到生成模型只能够制造数据而不是提供一个预测的密度函数时更是如此。

下面是一些学习生成模型的理由：

- 生成样本，这是最直接的理由。
- 训练并不包含最大似然估计。
- 由于生成器不会看到训练数据，过拟合的风险更低。
- GAN 十分擅长捕获模式的分布。

2.1.2 现实世界的一个比喻

我们来想象一下制作伪钞的犯罪嫌疑人和警察这个现实中的例子（见图 2-1）。我们来列举一下犯罪嫌疑人和警察对于钞票的关注点：

- 想要成为一名成功的伪钞制作者，犯罪嫌疑人需要蒙骗得了警察，使得警察无法区分出哪张是真实的钞票、哪张是伪造的钞票。
- 作为正义的一方，警察需要尽可能高效地发现哪些是假钞。

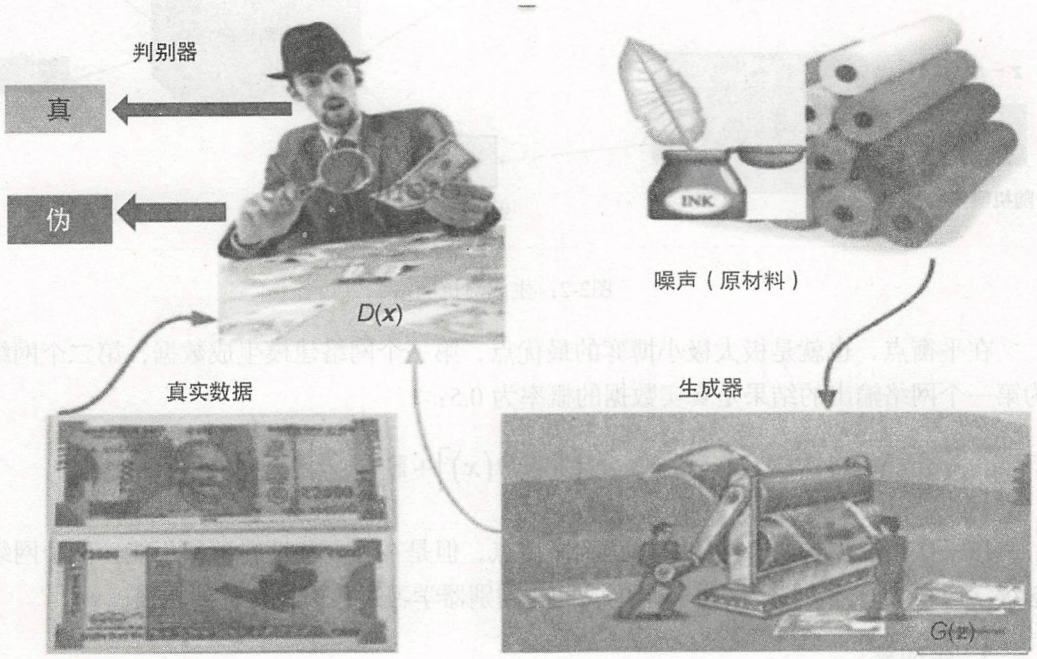


图2-1：GAN现实世界的比喻

这个场景可以建模为博弈论中的极大极小博弈，整个过程被称为**对抗性过程**（adversarial process）。GAN，由 Ian Goodfellow 于 2014 年在 *arXiv: 1406.2661* 中提出，它是一种两个神经网络相互竞争的特殊对抗过程。第一个网络生成数据，第二个网络试图区分真实数据与第一个网络创造出来的假数据。第二个网络会生成一个在[0,1]范围内的标量，代表数据是真实数据的概率。

2.1.3 GAN 的组成

在 GAN 中，第一个网络通常被称为生成器并且以 $G(z)$表示，第二个网络通常被称为判别器并且以 $D(x)$表示（见图 2-2）。

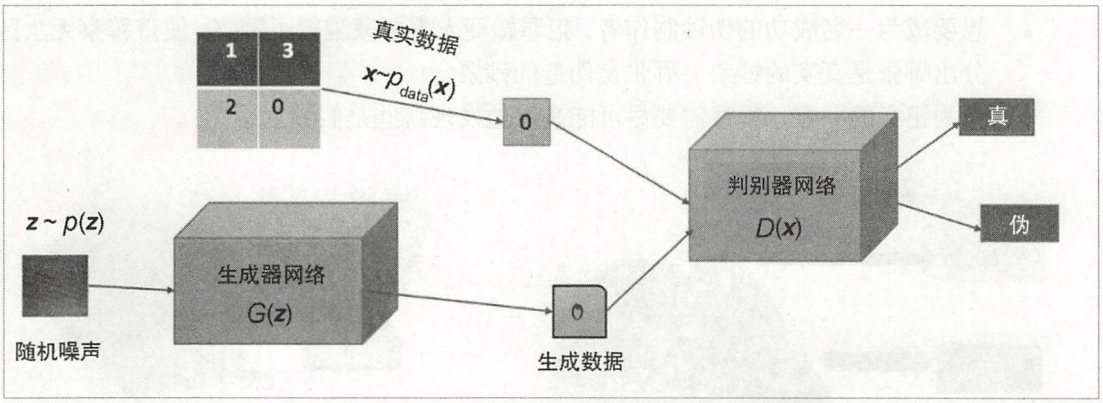


图2-2：生成对抗网络

在平衡点，也就是极大极小博弈的最优点，第一个网络建模生成数据，第二个网络认为第一个网络输出的结果是真实数据的概率为 0.5：

$$\min_G \max_D V(D,G) = \mathbb{E}_{x \sim p_{data}(x)} \left[\log D(x)\right] + \mathbb{E}_{x \sim p_z(z)} \left[\log\left(1 - D(G(z))\right)\right]$$

在一些情况下这两个网络可以达到平衡点，但是在另一些情况下却不能，两个网络会继续学习很长时间。图 2-3 是一个生成器和判别器学习过程损失变化的图表。

1. 生成器

生成器网络以随机的噪声作为输入并试图生成样本数据。在图 2-3 中，我们可以看到生成器 $G(z)$从概率分布 $p(z)$中接收输入 z，并且生成数据提供给判别器网络 $D(x)$。

2. 判别器

判别器网络以真实数据或者生成数据为输入，并试图预测当前输入是真实数据还是生成数据。其中一个输入 x 从真实的数据分布 $p_{data}(x)$中获取，接下来解决一个二分类问题并产生一个范围在 0 和 1 之间的标量。

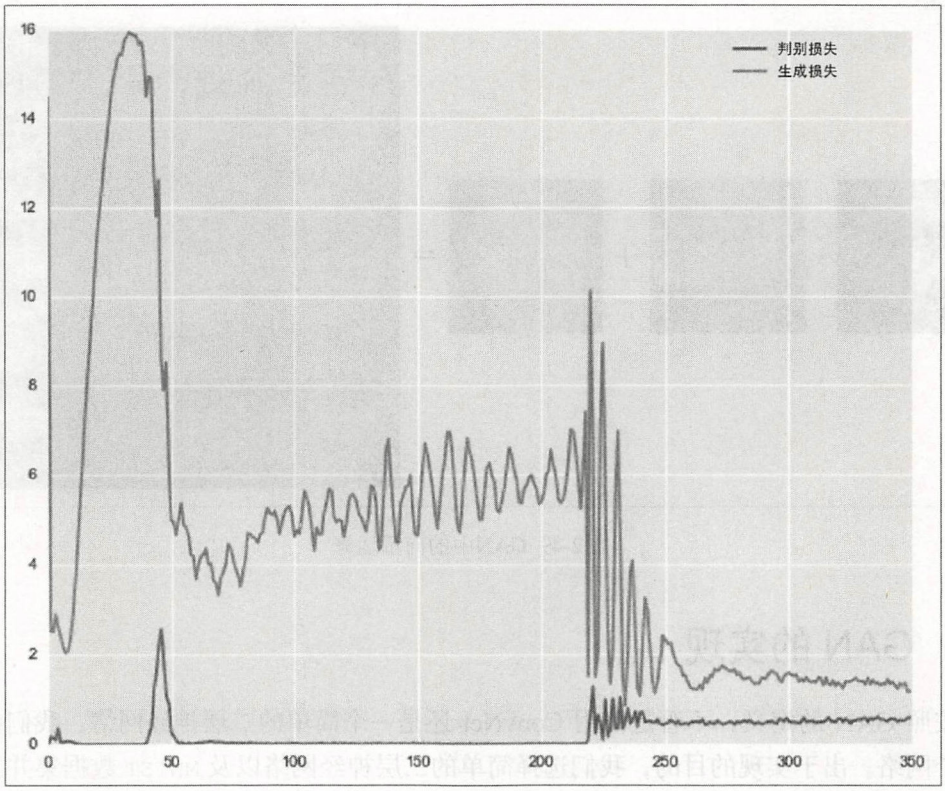


图2-3：生成器和判别器两个网络的损失

由于真实世界中无标记的数据量远远大于标记数据，而 GAN 十分擅长无监督学习任务，因此近些年来 GAN 变得越来越流行。它得以流行的另一个原因在于在多种生成模型中，GAN 可以生成最为逼真的图像。尽管这是一个很主观的评价，但这已经成为所有从业者的共识。

此外，GAN 还有着强大的表达能力：它可以在潜在空间（向量空间）内执行算数运算，并将其转换为对应特征空间内的运算。如图 2-4 所示，在潜在空间内有一张戴眼镜男人的照片，减去一个神经网络中男人的向量，再加上一个神经网络中女人的向量，最后会得到特征空间内一张戴眼镜女人的图像。这种表达能力的确令人震惊。

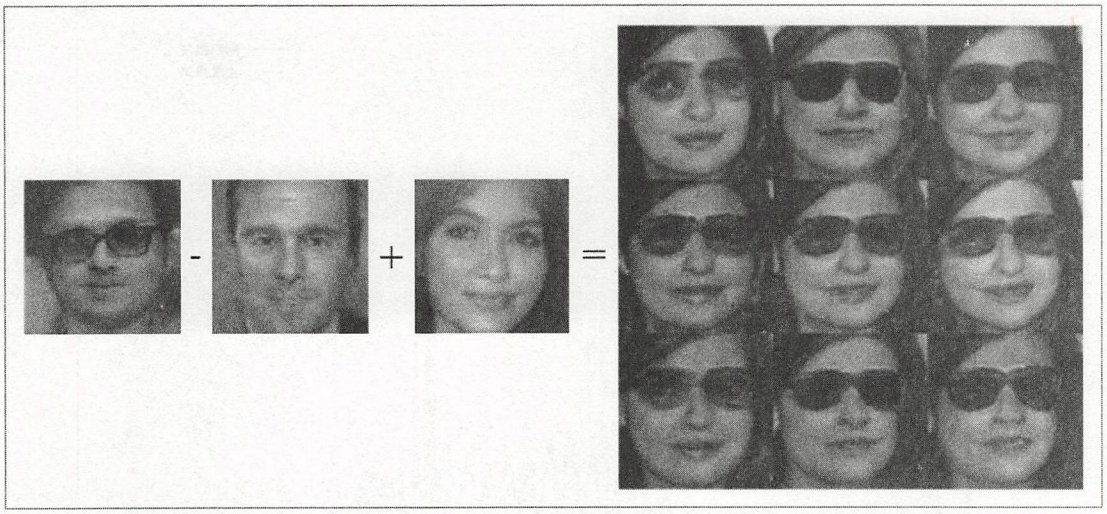


图2-4：GAN中的向量运算

2.2 GAN 的实现

按照 GAN 的定义，不管是使用 ConvNet 还是一个简单的二层神经网络，我们都需要有两个网络。出于实现的目的，我们选择简单的二层神经网络以及 MNIST 数据集并且使用 TensorFlow 来实现。MNIST 数据集由 28 像素 × 28 像素的手写字母灰度图组成。

```
# 为生成器生成随机噪声
Z = tf.placeholder(tf.float32, shape=[None, 100], name='Z')

# 生成器参数设置
G_W1 = tf.Variable(xavier_init([100, 128]), name='G_W1')
G_b1 = tf.Variable(tf.zeros(shape=[128]), name='G_b1')
G_W2 = tf.Variable(xavier_init([128, 784]), name='G_W2')
G_b2 = tf.Variable(tf.zeros(shape=[784]), name='G_b2')
theta_G = [G_W1, G_W2, G_b1, G_b2]


# 生成器网络
def generator(z):
    G_h1 = tf.nn.relu(tf.matmul(z, G_W1) + G_b1)
```

```
    G_log_prob = tf.matmul(G_h1, G_W2) + G_b2
    G_prob = tf.nn.sigmoid(G_log_prob)

    return G_prob
```

生成器 generator(z)从随机分布中（本例采用统一分布）接收一个 100 维向量为输入，并产生一个符合 MNIST 规范（28×28）的 786 维向量。这里的 z 是 $G(z)$的先验。通过这种方式可以学习到先验空间和 p_{data}（真实数据分布）空间之间的映射：

```
# 为判别器准备的MNIST图像输入设置
X = tf.placeholder(tf.float32, shape=[None, 784], name='X')

# 判别器参数设置
D_W1 = tf.Variable(xavier_init([784, 128]), name='D_W1')
D_b1 = tf.Variable(tf.zeros(shape=[128]), name='D_b1')
D_W2 = tf.Variable(xavier_init([128, 1]), name='D_W2')
D_b2 = tf.Variable(tf.zeros(shape=[1]), name='D_b2')
theta_D = [D_W1, D_W2, D_b1, D_b2]


# 判别器网络
def discriminator(x):
    D_h1 = tf.nn.relu(tf.matmul(x, D_W1) + D_b1)
    D_logit = tf.matmul(D_h1, D_W2) + D_b2
    D_prob = tf.nn.sigmoid(D_logit)

return D_prob, D_logit
```

判别器 discriminator(x)以 MNIST 图像作为输入并返回一个代表真实图像概率的标量。接下来让我们讨论一下训练 GAN 的算法。下面是论文 *arXiv: 1406.2661,2014* 中关于训练算法伪代码的内容（见图 2-5）。

for number of training iterations **do**
  for k steps **do**
    • Sample minibatch of m noise samples $\{\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
    • Sample minibatch of m examples $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ from data generating distribution $p_{\text{data}}(\boldsymbol{x})$.
    • Update the discriminator by ascending its stochastic gradient:

$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} \left[\log D\left(\boldsymbol{x}^{(i)}\right) + \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right) \right].$$

  end for
  • Sample minibatch of m noise samples $\{\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
  • Update the generator by descending its stochastic gradient:

$$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right).$$

end for

图2-5：GAN训练算法的伪代码

```
G_sample = generator(Z)

D_real, D_logit_real = discriminator(X)
D_fake, D_logit_fake = discriminator(G_sample)

# GAN原始论文中的损失函数
D_loss = -tf.reduce_mean(tf.log(D_real) + tf.log(1. - D_fake))
G_loss = -tf.reduce_mean(tf.log(D_fake))
```

TensorFlow 中的优化器只能做最小化，因此，为了最大化损失函数，我们在上面的代码中给损失加上了一个负号。与此同时，根据论文的伪代码算法，我们最好最大化 `tf.reduce_mean(tf.log(D_fake))` 而不是最小化 `tf.reduce_mean(1-tf.log(D_fake))`。接下来我们通过下面的过程一步步训练损失函数：

```
# 仅更新D(X)的参数，var_list=theta_D
D_solver = tf.train.AdamOptimizer().minimize(D_loss, var_list=theta_D)
# 仅更新G(X)的参数，var_list=theta_G
G_solver = tf.train.AdamOptimizer().minimize(G_loss, var_list=theta_G)
```

```
    def sample_Z(m, n):
        '''Uniform prior for G(Z)'''
        return np.random.uniform(-1., 1., size=[m, n])

    for it in range(1000000):
        X_mb, _ = mnist.train.next_batch(mb_size)

        _, D_loss_curr = sess.run([D_solver, D_loss], feed_dict={X: X_mb, Z:
sample_Z(mb_size, Z_dim)})
        _, G_loss_curr = sess.run([G_solver, G_loss], feed_dict={Z:
sample_Z(mb_size, Z_dim)})
```

我们以随机的噪声开始进行训练，G(Z)不断向 p_{data} 趋近。可以通过观察 G(Z)生成的样本和原始 MNIST 图像的区别来证明这件事。

图 2-6 是一些经过 60 000 次迭代后生成的结果。

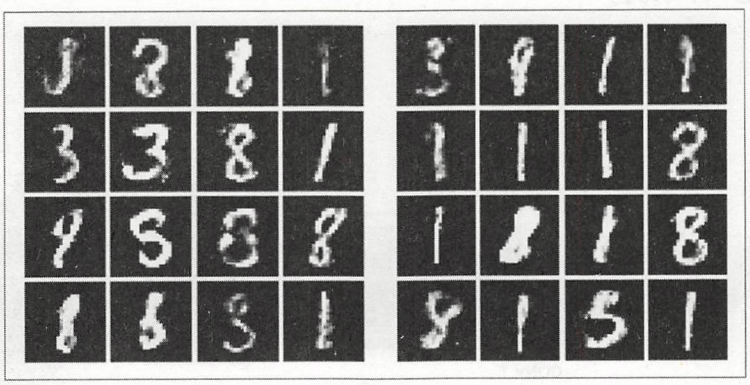


图2-6：GAN实现生成的输出图像

2.2.1 GAN 的应用

GAN 在多个领域都产生了许多让人振奋的结果。下面列举了近年来利用了 GAN 的一些令人激动的应用。

- 利用 CycleGAN 进行图像转换（将马变成斑马）以及利用条件 GAN 进行图像编辑。具体细节将在第 3 章中介绍。
- 利用 StackGAN 自动从文本中制作逼真的图像。利用探索式 GAN（Discovery GAN，DiscoGAN）进行图像风格的转换。具体细节将在第 4 章中介绍。

- 利用 SRGAN 通过预训练模型提升图像品质，制作高清晰度的图像。具体细节将在第 5 章中介绍。
- **通过特征生成逼真图像：**设想一个强盗闯入你的公寓，然而你并没有他的照片，现在警察局的系统可以根据你的描述生成强盗的照片并在数据库中进行搜索。更多的信息请参考 *arXiv: 1605.05396,2016*。
- 预测下一帧的视频或者动态生成视频：（`http://carlvondrick.com/tinyvideo/`）。

2.2.2 在 Keras 上利用 DCGAN 实现图像生成

在 Radford，L. Metz 和 S. Chintala 的论文 *arXiv: 1511.06434，2015* 利用深度卷积生成对抗网络进行无监督表征学习中第一次提及**深度卷积生成对抗网络（Deep Convolutional Generative Adversarial Network，DCGAN）**。

这个生成器将从一致分布空间中的 100 维度 z 通过一系列卷积操作映射到一个较小的空间，图 2-7 是一个示例。

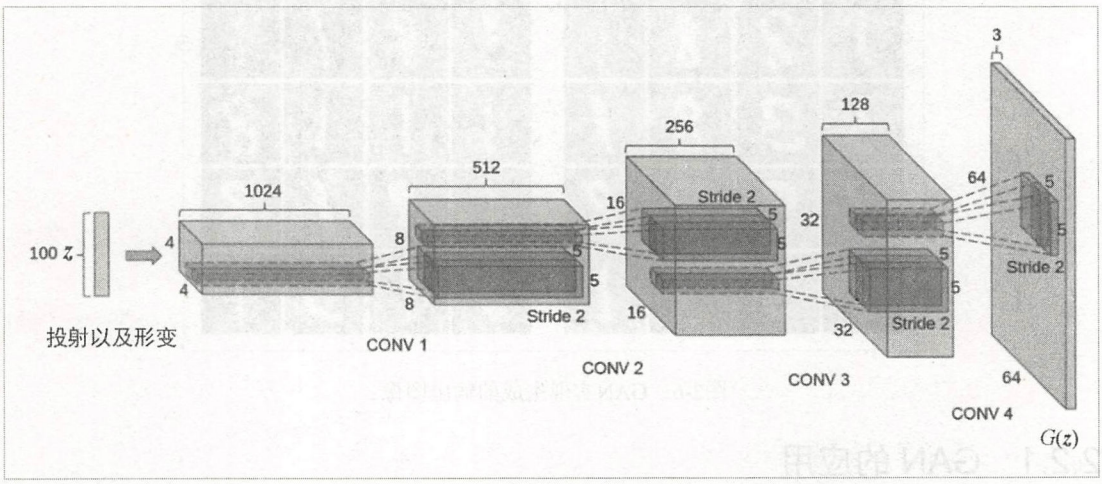


图2-7：DCGAN生成器的架构

（来源：*arXiv: 1511.06434,2015*）

DCGAN 通过下面一些架构性的约束来固化网络：

- 在判别器中使用步数卷积取代池化层，在生成器中使用小数步数卷积取代池化层。
- 在生成器和判别器中均使用批规范化。
- 消除架构中较深的全连接隐藏层，并且在最后只使用简单的平均值池化。

- 在生成器的输出层使用 tanh，在其他层均使用 ReLU 激发。
- 在判别器的所有层中都使用 Leaky ReLU 激发。

DCGAN 生成器在 Keras 上的实现可以参考下面链接中的代码 https://github.com/jacobgil/keras-dcgan。

通过下面的命令来启动模型的训练和生成（见图 2-8）：

```
python dcgan.py --mode train --batch_size <batch_size>
python dcgan.py --mode generate --batch_size <batch_size> --nice
```

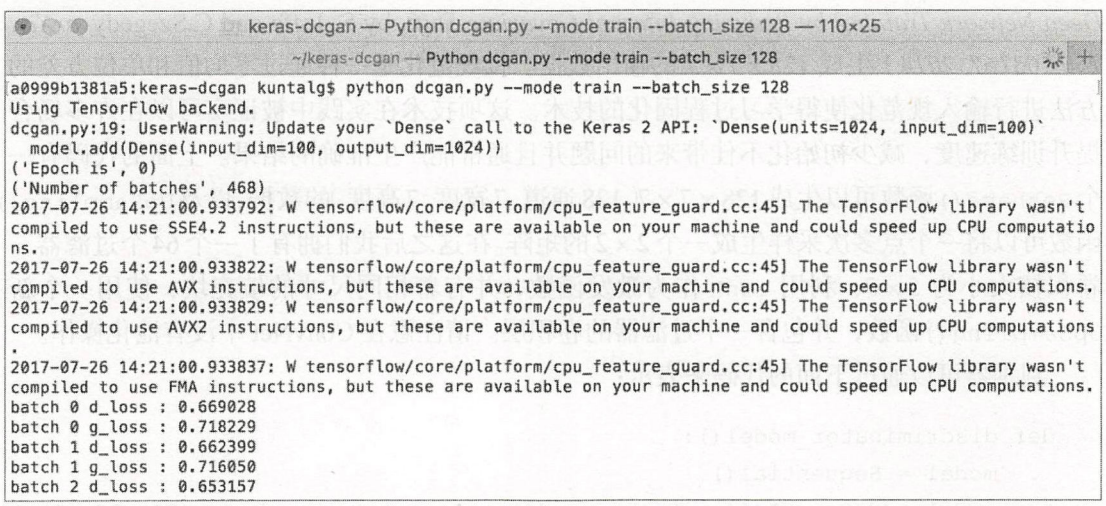


图2-8：模型的训练和生成

请注意上面打印出来的批次数字为根据所提供的输入图像形状和批大小计算所得。

现在让我们进入代码，生成器可以通过下面的代码来表示：

```
def generator_model():
    model = Sequential()
    model.add(Dense(input_dim=100, output_dim=1024))
    model.add(Activation('tanh'))
    model.add(Dense(128*7*7))
    model.add(BatchNormalization())
    model.add(Activation('tanh'))
    model.add(Reshape((7, 7, 128), input_shape=(128*7*7,)))
    model.add(UpSampling2D(size=(2, 2)))
    model.add(Conv2D(64, (5, 5), padding='same'))
```

```
    model.add(Activation('tanh'))
    model.add(UpSampling2D(size=(2, 2)))
    model.add(Conv2D(1, (5, 5), padding='same'))
    model.add(Activation('tanh'))
    return model
```

生成器的第一个全连接层接收一个 100 维的向量并且通过激发函数 tanh 产生一个 1024 维的输出。

网络中的下一个全连接层利用批规范化（参考论文 *Batch Normalization Accelerating Deep Network Training by Reducing Internal Covariate Shift*, by S. Ioffe and C.Szegedy, *arXiv: 1502.03167, 2014*）生成 128 × 7 × 7 的输出数据。批规范化是一种通过零均值和单位方差的方法进行输入规范化使得学习过程固化的技术。这项技术在实践中被证实可以在许多场合提升训练速度，减少初始化不佳带来的问题并且通常能产生准确的结果。上面的代码有一个 Reshape() 函数可以生成 128 × 7 × 7（128 通道，7 宽度，7 高度）的数据，以及 UpSampling() 函数可以将一个点多次采样生成一个 2 × 2 的矩阵。在这之后我们拥有了一个 64 个过滤器，卷积核大小为 5 × 5，利用 tanh 作为激发函数，并有着相同尺寸的填充块，使用一个新 UpSampling() 函数，并包含一个过滤器的卷积层。请注意在 ConvNet 中没有池化操作。

判别器可以通过下面的代码来表示：

```
def discriminator_model():
    model = Sequential()
    model.add(Conv2D(64, (5, 5), padding='same',input_shape=(28, 28, 1)))
    model.add(Activation('tanh'))
    model.add(MaxPooling2D(pool_size=(2, 2)))
    model.add(Conv2D(128, (5, 5)))
    model.add(Activation('tanh'))
    model.add(MaxPooling2D(pool_size=(2, 2)))
    model.add(Flatten())
    model.add(Dense(1024))
    model.add(Activation('tanh'))
    model.add(Dense(1))
    model.add(Activation('sigmoid'))
    return model
```

判别器以标准形状的（1, 28, 28）MNIST 图像为输入，应用 64 个 5 × 5 过滤器以 tanh 为激发函数的卷积。接下来执行一个 2 × 2 的最大值池化操作并紧跟一个卷积最大值池化操作。

最后两个阶段进行全连接操作，最后用来预测真伪的层只有一个 sigmoid 激发函数的神经元。在一段时间内，生成器和判别器都是通过 binary_crossentropy 作为损失（loss）函数来进行训练的。之后的每一个阶段，生成器产生一个对数字的预测（在本例中生成伪造的 MNIST 图像），判别器尝试在真实 MNIST 图像和生成图像的数据集中进行学习。经过一段时间，生成器就可以自动学会如何制作伪造的数字（见图 2-9）。

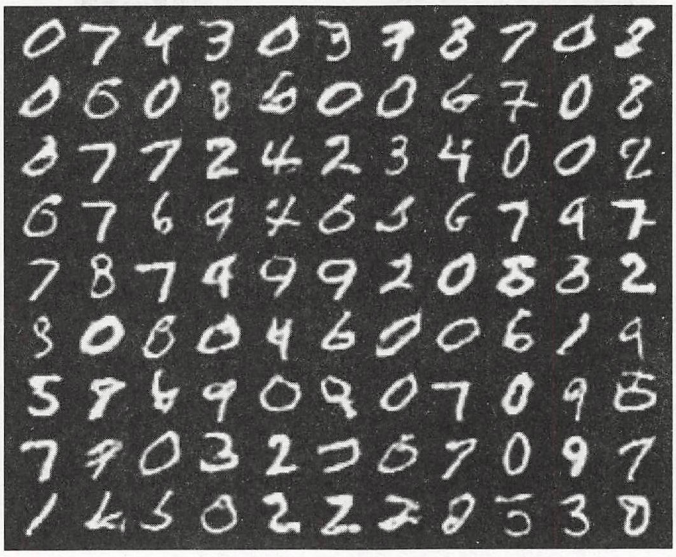


图2-9：深度卷积GAN产生的手写数字输出

请注意因为很难找到对抗双方的一个平衡点，所以训练 GAN 的过程会十分困难，本章的最后我们会介绍一些从业者所使用的宝贵技术和建议。

2.2.3 利用 TensorFlow 实现 SSGAN

半监督学习生成对抗网络（Semi-Supervised Learning Generative Adversarial Network，SSGAN）最初的动机是利用生成器生成的样本能力来提升图像分类任务的性能，进而提升判别器的泛化能力。其中关键的思想是将其中一个网络同时当作分类器和判别器进行训练（从真实图像中识别生成图像）。

对于一个包含 n 类数据的数据集，双训练（判别/分类）网络会以一个图像为输入，将真实图像分类到最初的 n 个类别中，将生成的图像分入第 n+1 个类别中，如图 2-10 所示。

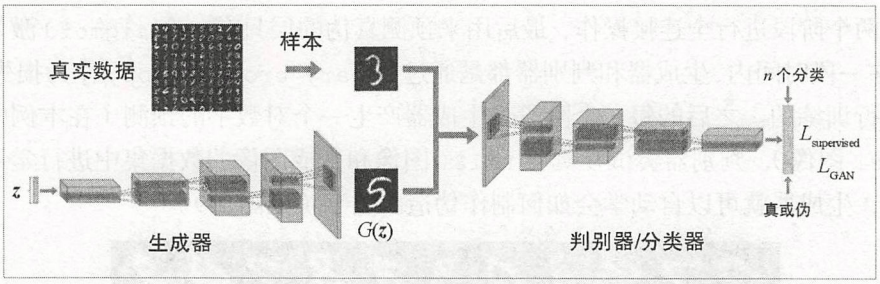


图2-10：SSGAN架构

（来源：https://github.com/gitlimlab/SSGAN-Tensorflow/blob/master/figure/ssgan.png）

多任务联合学习的框架包含两个损失，首先是监督学习的损失：

$$\mathcal{L}_{\text{supervised}} = -\mathbb{E}_{x,y\sim p_{\text{data}}(x,y)} \log p_{\text{model}}\left(y \mid x, y < n+1\right)$$

其次是 GAN 判别器的损失：

$$\mathcal{L}_{\text{GAN}} = -\left\{\mathbb{E}_{x\sim p_{\text{data}}(x)} \log\left[1 - p_{\text{model}}\left(y = n+1 \mid x\right)\right] + \mathbb{E}_{x\sim \text{Generator}} \log p_{\text{model}}\left(y = n+1 \mid x\right)\right\}$$

在训练过程中这两个损失会同时最小化。

安装环境

执行下面的步骤，在 CIFAR-10 数据集上执行 SSGAN：

1. 克隆下面 git 仓库的代码：https://github.com/gitlimlab/SSGAN-Tensorflow（见图 2-11）。

```
a0999b1381a5:~ kuntalg$ git clone https://github.com/gitlimlab/SSGAN-Tensorflow.git
Cloning into 'SSGAN-Tensorflow'...
remote: Counting objects: 494, done.
remote: Total 494 (delta 0), reused 0 (delta 0), pack-reused 494
Receiving objects: 100% (494/494), 50.86 MiB | 1.26 MiB/s, done.
Resolving deltas: 100% (295/295), done.
```

图2-11：克隆代码

2. 更改当前目录：

```
cd SSGAN-Tensorflow/
```

3. 下载 CIFAR-10 数据集（见图 2-12）。

```
a0999b1381a5:SSGAN-Tensorflow kuntalg$ python download.py --dataset CIFAR10
./datasets/cifar10/cifar-10-python.tar.gz
Downloading CIFAR10
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100  162M  100  162M    0     0  1248k      0  0:02:13  0:02:13 --:--:-- 1757k
Preprocessing data...
[==================================================================================================] 100%
```

图2-12：下载数据集

4. 训练模型（见图 2-13）。

```
SSGAN-Tensorflow — Python trainer.py --dataset CIFAR10 — 110×27
~/keras-dcgan — -bash   ~/keras-dcgan — -bash   ~/Desktop — -bash   Python train...aset CIFAR10
a0999b1381a5:SSGAN-Tensorflow kuntalg$ python trainer.py --dataset CIFAR10
[2017-07-27 12:43:49,061] Reading ./datasets/cifar10/data.hy ...
[2017-07-27 12:43:49,070] Reading Done: ./datasets/cifar10/data.hy
[2017-07-27 12:43:49,072] Reading ./datasets/cifar10/data.hy ...
[2017-07-27 12:43:49,072] Reading Done: ./datasets/cifar10/data.hy
[2017-07-27 12:43:49,073] Train Dir: ./train_dir/default-CIFAR10_lr_0.0001_update_G5_D1-20170727-124349
[2017-07-27 12:43:49,073] input_ops [inputs]: Using 50000 IDs from dataset
[2017-07-27 12:43:49,395] input_ops [inputs]: Using 10000 IDs from dataset
Generator
Generator Tensor("Generator/g_1_deconv/Relu:0", shape=(64, 2, 2, 384), dtype=float32)
Generator Tensor("Generator/g_2_deconv/Relu:0", shape=(64, 6, 6, 128), dtype=float32)
Generator Tensor("Generator/g_3_deconv/Relu:0", shape=(64, 14, 14, 64), dtype=float32)
Generator Tensor("Generator/g_4_deconv/Tanh:0", shape=(64, 32, 32, 3), dtype=float32)
Discriminator
Discriminator Tensor("d_1_conv/dropout/mul:0", shape=(64, 16, 16, 64), dtype=float32)
Discriminator Tensor("d_2_conv/dropout/mul:0", shape=(64, 8, 8, 128), dtype=float32)
Discriminator Tensor("d_3_conv/dropout/mul:0", shape=(64, 4, 4, 256), dtype=float32)
Discriminator Tensor("Discriminator/d_4_fc/BiasAdd:0", shape=(64, 11), dtype=float32)
Successfully loaded the model.
[2017-07-27 12:43:50,865] ********* d_var **********
---------
Variables: name (type shape) [size]
---------
Discriminator/d_1_conv/w:0 (float32_ref 5x5x3x64) [4800, bytes: 19200]
Discriminator/d_1_conv/biases:0 (float32_ref 64) [64, bytes: 256]
Discriminator/d_1_conv/BatchNorm/beta:0 (float32_ref 64) [64, bytes: 256]
```

图2-13：训练模型

5. 测试模型效果：

```
python evaler.py --dataset CIFAR10 --checkpoint ckpt_dir
```

下面让我们进入代码，生成器从下面的一致分布中获取噪声。

```
z = tf.random_uniform([self.batch_size, n_z], minval=-1, maxval=1,
dtype=tf.float32)
```

接下来生成器模型利用 `reshape` 方法将噪声扁平化成一维向量。然后使用三层利用 `ReLU` 为激发函数的对输入层的逆卷积，紧接着使用一层利用 `tanh` 为激发函数的卷积层生成对应维度（h=高度；w=宽度；c=通道数，灰度图为 1，彩色图为 3）的输出图像。

```
    # 生成器模型函数
        def G(z, scope='Generator'):
            with tf.variable_scope(scope) as scope:
                print ('\033[93m'+scope.name+'\033[0m')
                z = tf.reshape(z, [self.batch_size, 1, 1, -1])
                g_1 = deconv2d(z, deconv_info[0], is_train, name='g_1_deconv')
                print (scope.name, g_1)
                g_2 = deconv2d(g_1, deconv_info[1], is_train,
name='g_2_deconv')
                print (scope.name, g_2)
                g_3 = deconv2d(g_2, deconv_info[2], is_train,
name='g_3_deconv')
                print (scope.name, g_3)
                g_4 = deconv2d(g_3, deconv_info[3], is_train,
name='g_4_deconv', activation_fn='tanh')
                print (scope.name, g_4)
                output = g_4
                assert output.get_shape().as_list() == self.image.get_
shape().as_list(), output.get_shape().as_list()
            return output

    # 逆卷积函数
    def deconv2d(input, deconv_info, is_train, name="deconv2d",
stddev=0.02,activation_fn='relu'):
        with tf.variable_scope(name):
            output_shape = deconv_info[0]
            k = deconv_info[1]
            s = deconv_info[2]
            deconv = layers.conv2d_transpose(input,
                num_outputs=output_shape,
                weights_initializer=tf.truncated_normal_initializer
(stddev=stddev),
                biases_initializer=tf.zeros_initializer(),
                kernel_size=[k, k], stride=[s, s], padding='VALID')
            if activation_fn == 'relu':
                deconv = tf.nn.relu(deconv)
```

```
            bn = tf.contrib.layers.batch_norm(deconv, center=True,
scale=True,
                decay=0.9, is_training=is_train, updates_collections=None)
        elif activation_fn == 'tanh':
            deconv = tf.nn.tanh(deconv)
        else:
            raise ValueError('Invalid activation function.')
        return deconv
```

这个判别器以图像为输入，并尝试将结果进行 n+1 分类。其中的一些卷积层使用了 Leaky ReLU 以及批规范化，接下来使用了对输入图像的丢弃，最终利用 `softmax` 函数对结果进行分类：

```
    # 判别器模型函数
        def D(img, scope='Discriminator', reuse=True):
            with tf.variable_scope(scope, reuse=reuse) as scope:
                if not reuse: print ('\033[93m'+scope.name+'\033[0m')
                d_1 = conv2d(img, conv_info[0], is_train, name='d_1_conv')
                d_1 = slim.dropout(d_1, keep_prob=0.5, is_training=is_train,
scope='d_1_conv/')
                if not reuse: print (scope.name, d_1)
                d_2 = conv2d(d_1, conv_info[1], is_train, name='d_2_conv')
                d_2 = slim.dropout(d_2, keep_prob=0.5, is_training=is_train,
scope='d_2_conv/')
                if not reuse: print (scope.name, d_2)
                d_3 = conv2d(d_2, conv_info[2], is_train, name='d_3_conv')
                d_3 = slim.dropout(d_3, keep_prob=0.5, is_training=is_train,
scope='d_3_conv/')
                if not reuse: print (scope.name, d_3)
                d_4 = slim.fully_connected(
                    tf.reshape(d_3, [self.batch_size, -1]), n+1,
scope='d_4_fc', activation_fn=None)
                if not reuse: print (scope.name, d_4)
                output = d_4
                assert output.get_shape().as_list() == [self.batch_size, n+1]
                return tf.nn.softmax(output), output
```

```
    # 带丢弃的卷积函数
    def conv2d(input, output_shape, is_train, k_h=5, k_w=5, stddev=0.02,
name="conv2d"):
        with tf.variable_scope(name):
            w = tf.get_variable('w', [k_h, k_w, input.get_shape()[-1],
output_shape],
                    initializer=tf.truncated_normal_initializer(stddev=stddev))
            conv = tf.nn.conv2d(input, w, strides=[1, 2, 2, 1], padding='SAME')

            biases = tf.get_variable('biases', [output_shape],
initializer=tf.constant_initializer(0.0))
            conv = lrelu(tf.reshape(tf.nn.bias_add(conv, biases),
conv.get_shape()))
            bn = tf.contrib.layers.batch_norm(conv, center=True, scale=True,
                decay=0.9, is_training=is_train, updates_collections=None)
        return bn

    # Leaky Relu函数
    def lrelu(x, leak=0.2, name="lrelu"):
      with tf.variable_scope(name):
      f1 = 0.5 * (1 + leak)
      f2 = 0.5 * (1 - leak)
    return f1 * x + f2 * abs(x)
```

判别器网络有两个损失函数，一个（s_loss）利用 Huber 损失函数（相较于平方错误损失，Huber 损失在异常点情况下更加健壮）计算监督学习分类下对 CIFAR-10 图像分类的损失，另一个损失（d_loss）用来表示对生成图像进行真伪分类的损失，这个损失函数通过 softmax 和交叉熵来计算（见图 2-14 和图 2-15）：

```
    # 判别器/分类器损失
    s_loss = tf.reduce_mean(huber_loss(label, d_real[:, :-1]))
```

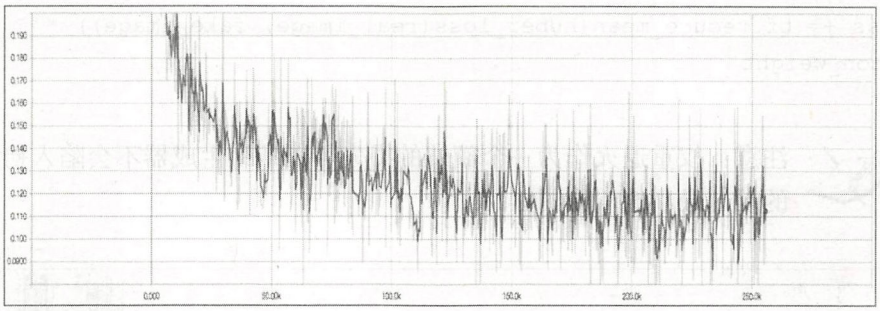


图2-14：监控判别器的损失

```
    d_loss_real = tf.nn.softmax_cross_entropy_with_logits(logits=d_real_
logits, labels=real_label)
    d_loss_fake = tf.nn.softmax_cross_entropy_with_logits(logits=d_fake_
logits, labels=fake_label)
    d_loss = tf.reduce_mean(d_loss_real + d_loss_fake)
```

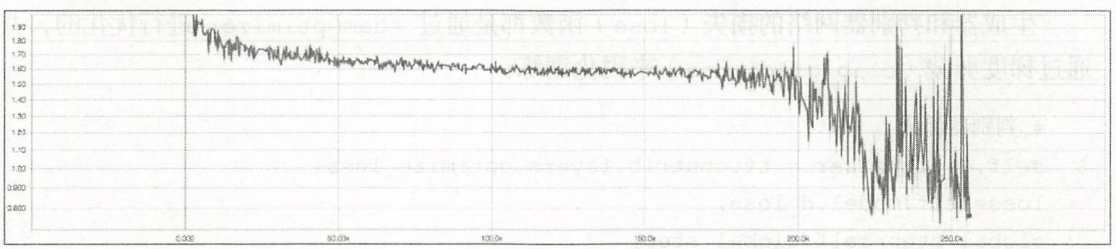


图2-15：判别器的总损失（真+伪损失）

```
    # Huber损失
    def huber_loss(labels, predictions, delta=1.0):
        residual = tf.abs(predictions - labels)
        condition = tf.less(residual, delta)
        small_res = 0.5 * tf.square(residual)
        large_res = delta * residual - 0.5 * tf.square(delta)
        return tf.where(condition, small_res, large_res)


    # 生成器损失
    g_loss = tf.reduce_mean(tf.log(d_fake[:, -1]))
```

```
    g_loss += tf.reduce_mean(huber_loss(real_image, fake_image)) *
self.recon_weight
```

注意：权重退火作为一个额外的损失可以帮助生成器不会陷入初始的局部最小值（见图 2-16）。

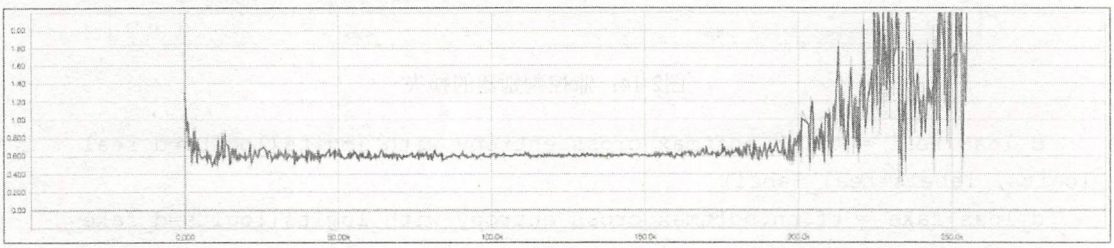


图2-16：生成器损失

生成器和判别器网络的损失（loss）函数都是通过 AdamOptimizer 进行优化的，并通过梯度剪裁（clip_gradients）来固化训练：

```
    # 判别器优化
    self.d_optimizer = tf.contrib.layers.optimize_loss(
    loss=self.model.d_loss,
    global_step=self.global_step,
    learning_rate=self.learning_rate*0.5,
    optimizer=tf.train.AdamOptimizer(beta1=0.5),
    clip_gradients=20.0,
    name='d_optimize_loss',
    variables=d_var
    )


    # 生成器优化
    self.g_optimizer = tf.contrib.layers.optimize_loss(
    loss=self.model.g_loss,
    global_step=self.global_step,
    learning_rate=self.learning_rate,
    optimizer=tf.train.AdamOptimizer(beta1=0.5),
    clip_gradients=20.0,
```

```
    name='g_optimize_loss',
    variables=g_var
    )
```

最终监督学习的损失（s_loss）和生成对抗的损失（包含判别器损失 d_loss 和生成器损失 g_loss）被联合训练，以最终达到总损失的最小化。

```
    for s in xrange(max_steps):
                step, accuracy, summary, d_loss, g_loss, s_loss, step_time,
prediction_train, gt_train, g_img = \
                 self.run_single_step(self.batch_train, step=s, is_train=True)
```

经过 150 个阶段训练后的生成样本如图 2-17 所示。

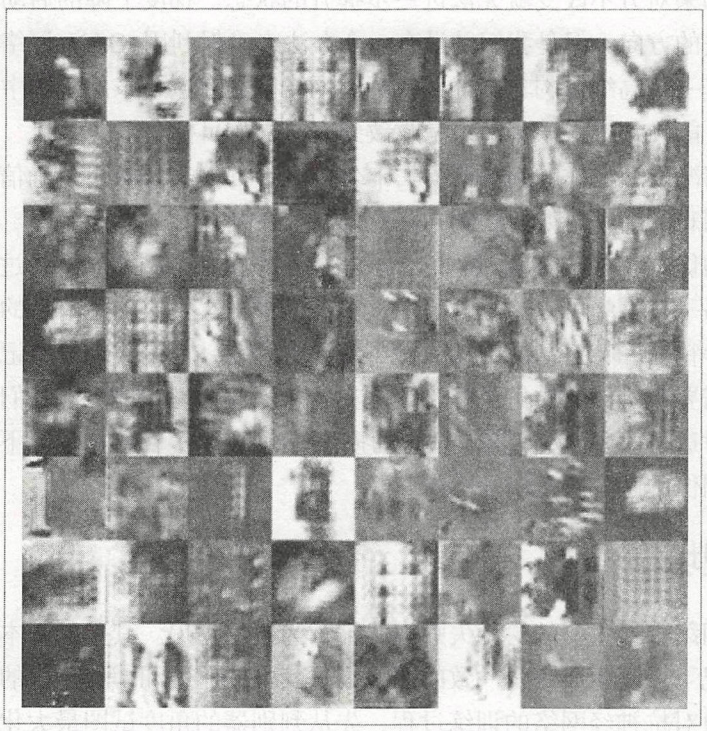


图2-17：生成样本

2.3 GAN 模型的挑战

训练 GAN 本质上是生成器网络 $G(z)$和判别器网络 $D(z)$相互竞争并达到最优，更确切地说是达到纳什均衡的一个过程。根据维基百科的定义，纳什均衡是一个经济学和博弈论的术语，代表着一个系统的稳定状态，此时系统的各个参与者没有一个人可以通过独自改变行动而增加收益。

2.3.1 启动及初始化的问题

如果你仔细思考上面的定义，会发现这正是 GAN 试图做的事情；生成器和判别器最终达到了一个如果对方不改变就无法进一步提升的状态。梯度下降的启动会选择一个减小所定义问题损失的方向，但是我们并没有一个办法来确保利用 GAN 网络可以进入纳什均衡的状态，这是一个高维度的非凸优化目标。网络试图在接下来的步骤中最小化非凸优化目标，最终有可能导致进入振荡而不是收敛到底层真实目标。

在大多数情况下，当你的判别器损失十分接近于 0 时，那么可以确信你的模型出现了问题，然而更困难的问题是如何找到问题究竟出现在哪里。

一个常见的优化 GAN 训练过程的手段是故意停止某个网络的学习过程，或者降低学习速率，使得另一个网络可以追上来。在大多数场景下，生成器是落后的一方，我们需要让判别器进行等待。在特定情况下这样做可能没有问题，但是我们需要记住想要让生成器的质量更好，需要判别器的质量更好，反之亦然。因此，理想情况下我们希望两个网络以同样的速率同时进行改善。判别器最理想的损失接近于 0.5，在这个情况下对于判别器来说其无法从真实图像中区分出生成的图像。

2.3.2 模型坍塌

生成对抗网络中最主要的一种失败模式被称为模型坍塌。其中的基本原理是生成器可能会在某种情况下重复生成完全一致的图像，这其中的原因和博弈论中的启动相关。我们可以这样来想象对抗神经网络的训练过程，先从判别器的角度试图最大化，再从生成器的角度试图最小化。如果生成器最小化开始之前，判别器已经完全最大化，所有工作还可以正常运行。然而如果我们通过另一种方式首先尝试最小化生成器，接下来尝试最大化判别器，那么工作就无法正常运行。原因在于如果我们保持判别器不变，它会将空间中的某些点标记为最有可能是真的而不是假的，这样生成器就会选择将所有的噪声输入映射到那些

最可能为真的点。

2.3.3 计数方面的问题

GAN 在某些情况下有太多的视角，并错误地判断了物体在特定位置应该出现的数量，如图 2-18 所示。和原始图像相比，GAN 在动物头上生成了过多数量的眼睛。

图2-18：计数方面的问题

（来源：*NIPS 2016- arXiv: 1701.00160, 2017*）

2.3.4 角度方面的问题

GAN 通常不能很好地区分图像是从前方观测的结果还是从后方观测的结果，因此在通过 3D 物体生成 2D 表现形式时通常会出现问题，如图 2-19 所示。

图2-19：角度方面的问题

（来源：*NIPS 2016- arXiv: 1701.00160, 2017*）

2.3.5 全局结构方面的问题

和角度方面的问题类似，GAN 无法理解全局方面的结构。如图 2-20 左下方的图像，GAN 生成了一个四重的奶牛，这只奶牛用 4 只后腿站立起来，这在现实中是绝对不会出现的。

图2-20：全局结构方面的问题

（来源：*NIPS 2016- arXiv: 1701.00160, 2017*）

2.4 提升 GAN 训练效果的方法

为了克服训练 GAN 模型中的问题，深度学习的从业者根据问题的不同特性发明了各种技巧，下面将会介绍其中的一些方法。

2.4.1 特征匹配

GAN 不稳定的特点可以通过给生成器一个新的训练目标来解决，这种方式也可以避免生成器过拟合当前的判别器。

这种方法的思想是使用判别器中间层的特征来匹配图像的真伪，并将其作为一个监督

信号来训练生成器。

通过这种方式，我们训练的生成器的生成数据会匹配真实数据的统计特性以及判别器中间层的预期特征值。在判别器的训练过程中，我们让判别器去寻找那些最能很好地判别真实数据的特征而不是那些由当前模型生成数据的特征。

2.4.2 小批量

模型坍塌的问题可以通过给判别器加入一些额外特征的方法来解决。这样判别器每次考虑一小批的样本而不是一个单独样本。如果这些特征可以表示样本之间的距离，那么我们很容易就可以检测出当前的生成器是不是已经开始坍塌，而不是继续鼓励每一个生成器生成的样本继续向单个最大似然点靠近。小批量整体来说表现更接近实际，而且可以保证不同样本之间在空间上有合适的距离。

2.4.3 历史平均

历史平均的思想是加入一个惩罚项来惩罚那些和历史平均权重相差过多的权重值，具体来说，代价函数是当前参数值和历史上最近 t 批该参数平均值的距离。

```
distance (current parameters, average of parameters over the last t
batches)
```

2.4.4 单侧标签平滑

通常情况下我们使用标签 0 代表图像是真实的，1 代表图像是伪造的。我们还可以使用一些更平滑的标签，例如 0.1 和 0.9，它们可以使得网络在一些对抗的例子中更加健壮。

2.4.5 输入规范化

大多数情况下最好将图像规范化至-1 到 1 的范围之间并使用 tanh 作为生成器最后一层的激发函数。

2.4.6 批规范化

核心思想是针对真实数据和伪造数据构建不同的小批，每一个小批只能全部是真实图像或者全部是生成图像。当批规范化无法实现时，你可以使用实例规范化（对每个样本减去平均值再除以标准差）。

2.4.7 利用 ReLU 和 MaxPool 避免稀疏梯度

如果梯度稀疏，GAN 博弈的稳定性会受到很大影响，Leaky ReLU 对生成器和判别器都会有很多帮助。

对于向下采样的场景组合平均值池化，`Conv2d` 和 `stride` 在向上采样的场景中组合 `PixelShuffle`、`ConvTranspose2d` 和 `stride`：

```
PixelShuffle- arXiv: 1609.05158, 2016
```

2.4.8 优化器和噪声

针对生成器使用 ADAM 作为优化器，针对判别器使用 SGD 作为优化器，并且通过在生成器的不同层中去除输入来作为噪声的来源。

2.4.9 不要仅根据统计信息平衡损失

与其遵守原则性的方法不如凭借直觉：

```
while lossD > A:
  train D
while lossG > B:
  train G
```

注意：尽管书中提供了很多提升训练效果的方法，但对抗生成模型在深度学习和 AI 领域依然是一个较新的方向。和其他高速发展的领域类似，它也有许多需要改进的地方。

2.5 总结

到现在为止，你应该已经了解了如何通过 GAN 的概念将深度学习引入无监督学习的领域。你已经利用 `MNIST` 和 `CIFAR` 数据集生成了一些逼真的手写数字、飞机、汽车、鸟的图像。同时，你也了解了和生成对抗网络相关的挑战以及应对它们的一些实用调优技巧。

在接下来的几章中我们会继续自己的征程，了解几个基于 GAN 体系的变种以及如何通过真实数据来完成一系列令人惊叹的任务。

3 图像风格跨域转换

生成对抗网络是目前深度学习领域中发展最为快速的一个分支，它可以用于诸如图像编辑和着色、风格转换、物体变形、照片增强等多个领域。

在本章中，我们首先介绍根据特定条件或者特性来生成或者编辑图像的技术。接下来我们会应用**边界均衡**（Boundary Equilibrium）的方法来衡量模型的收敛程度，以及固化 GAN 训练过程，避免模型坍塌问题。最后我们将会利用**循环一致生成网络**（Cycle Consistent Generative Network）进行图像风格的跨域转换，将苹果变成橘子或者将马变成斑马。

本章将会包含以下内容：

- 什么是 CGAN，它的概念以及架构。
- 利用 CGAN 从 `Fashion-MNIST` 数据集中生成时尚衣柜。
- 利用边界均衡和沃瑟斯坦距离固化 GAN 训练。
- 利用 CycleGAN 实现图像风格转换。
- 利用 TensorFlow 将苹果变成橘子。
- 自动化将图像中的马变成斑马。

3.1 弥补监督学习和无监督学习之间的空隙

人类通过观察和体验物理世界来学习，我们的大脑十分擅长预测，不需要显式地经过复杂计算就可以得到正确的答案。监督学习的过程就是预测数据和标签之间的关联关系，

目标是对于未曾见过的数据也能有很好的预测能力。在非监督学习中，数据并没有被标记，目标通常也不是对新数据产生某种预测。

在现实世界中标记数据是十分稀有和昂贵的。生成对抗网络通过生成伪造的/合成的数据并尝试判断生成样本真伪的方法学习，这相当于采用了监督学习的方法来做无监督的学习任务。做分类任务的判别器在这里是一个监督学习的组件，但是 GAN 的最终目标是了解真实数据的模样，即对真实数据的分布或者密度进行预估，并能够根据学到的知识生成新的数据。

3.2 条件 GAN 介绍

一个**生成对抗网络（GAN）**会同时训练两个网络——一个从未知分布或者噪声中学习生成伪造样本的生成器，以及一个学习如何从样本中区分真伪的判别器。

在条件 **GAN（Conditional GAN，CGAN）**中，生成器并不是从一个未知的噪声分布开始学习，而是通过一个特定的条件或者某些特征（例如一个图像的标签或其他更细节的一些特征）开始学习如何生成伪造样本。现在我们给生成器和判别器都加入一些条件变量，即给两个网络都加入一个参数向量 $\boldsymbol{y}$。这样判别器 $D(\boldsymbol{x},\boldsymbol{y})$和生成器 $G(\boldsymbol{z},\boldsymbol{y})$都有了一组联合条件变量。

现在 CGAN 的目标函数变成了：

$$\min_G \max_D V(D,G) = \mathbb{E}_{x\sim p_{\text{data}}(x)}\left[\log D(x)\right] + \mathbb{E}_{z\sim p_z(z)}\left[\log\left(1-D\left(G(z)\right)\right)\right] \Longrightarrow \text{GAN}$$

$$\min_G \max_D V(D,G) = \mathbb{E}_{x\sim p_{\text{data}}(x)}\left[\log D(x)\right] + \mathbb{E}_{z\sim p_z(z)}\left[\log\left(1-D\left(G(z,y),y\right)\right)\right] \Longrightarrow \text{CGAN}$$

GAN 和 CGAN 的损失函数区别在于判别器和生成器多出来一个参数 $\boldsymbol{y}$。从图 3-1 所示的架构中可以看出，CGAN 相比于 GAN 增加了一个输入层条件向量 $\boldsymbol{C}$，同时连接到了判别器网络和生成器网络（见图 3-1）。

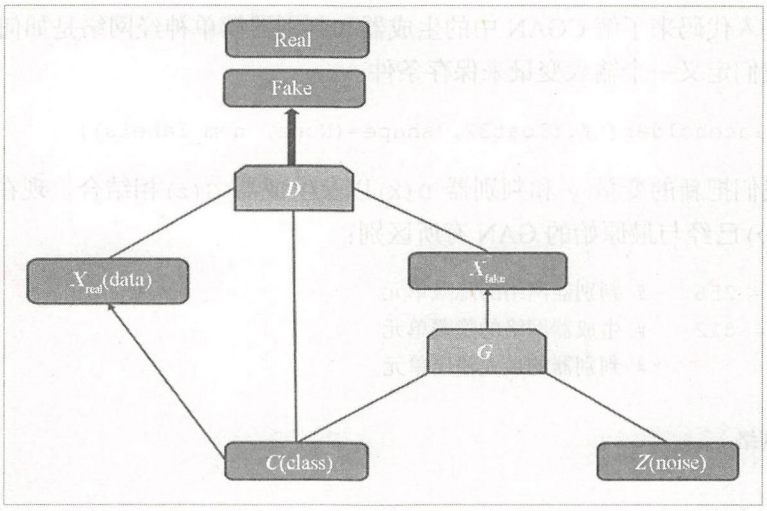
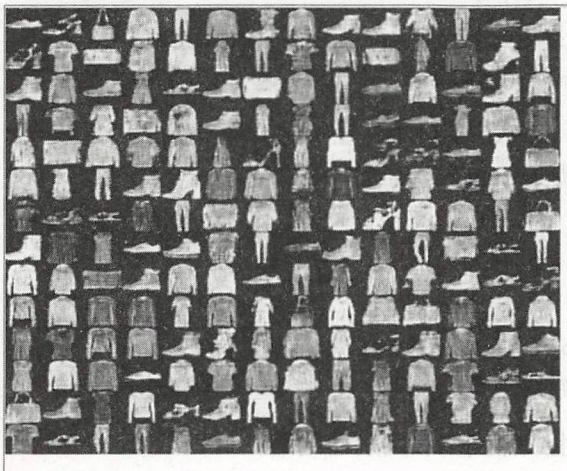


图3-1：CGAN架构

3.2.1 利用 CGAN 生成时尚衣柜

在本例中，我们将会利用 Fashion-MNIST 数据集（https://github.com/zalandoresearch/fashion-mnist）通过条件 GAN 来生成一个时尚衣柜。Fashion-MNIST 数据集和最初的 MNIST 数据集类似，但多了一些加标签的灰度图（见图 3-2）。



Label	Description
0	T-shirt/top
1	Trouser
2	Pullover
3	Dress
4	Coat
5	Sandal
6	Shirt
7	Sneaker
8	Bag
9	Ankle boot

时尚衣柜-图像　　时尚衣柜-标签

图3-2：衣柜图像及标签

让我们进入代码来了解 CGAN 中的生成器和判别器简单神经网络是如何工作的。

首先，我们定义一个输入变量来保存条件：

```
Y = tf.placeholder(tf.float32, shape=(None, num_labels))
```

其次，我们把新的变量 y 和判别器 D(X) 以及生成器 G(z) 相结合。现在判别器(x,y)和生成器(z,y)已经与最原始的 GAN 有所区别：

```
Dhidden = 256    # 判别器网络的隐藏单元
Ghidden = 512    # 生成器网络的隐藏单元
K = 8            # 判别器的最大输出单元

# 判别器网络

def discriminator(x, y):
    u = tf.reshape(tf.matmul(x, DW1x) + tf.matmul(y, DW1y) + Db1, [-1, K,
Dhidden])
    Dh1 = tf.nn.dropout(tf.reduce_max(u, reduction_indices=[1]), keep_prob)
    return tf.nn.sigmoid(tf.matmul(Dh1, DW2) + Db2)

# 生成器网络

def generator(z,y):
    Gh1 = tf.nn.relu(tf.matmul(Z, GW1z) + tf.matmul(Y, GW1y) + Gb1)
    G = tf.nn.sigmoid(tf.matmul(Gh1, GW2) + Gb2)
    return G
```

然后，我们使用新的网络并定义损失（loss）函数：

```
G_sample = generator(Z, Y)
DG = discriminator(G_sample, Y)

Dloss = -tf.reduce_mean(tf.log(discriminator(X, Y)) + tf.log(1 - DG))
Gloss = tf.reduce_mean(tf.log(1 - DG) - tf.log(DG + 1e-9))
```

在训练过程中，我们将 y 输入给生成器网络和判别器网络：

```
X_mb, y_mb = mnist.train.next_batch(mini_batch_size)
```

```
    Z_sample = sample_Z(mini_batch_size, noise_dim)

    _, D_loss_curr = sess.run([Doptimizer, Dloss], feed_dict={X: X_mb, Z:
Z_sample, Y:y_mb, keep_prob:0.5})

    _, G_loss_curr = sess.run([Goptimizer, Gloss], feed_dict={Z: Z_sample,
Y:y_mb, keep_prob:1.0})
```

最后，我们基于特定条件生成了新的数据样本。在本例中，我们使用图像标签作为条件，并选择标签值为 7，也就是生成运动鞋（Sneaker）的图像。条件变量 y_sample 是一个单热编码的向量，只在索引为 7 的位置是 1，其他位置都是 0：

```
nsamples=6

    Z_sample = sample_Z(nsamples, noise_dim)
      y_sample = np.zeros(shape=[nsamples, num_labels])
      y_sample[:, 7] = 1 # generating image based on label

      samples = sess.run(G_sample, feed_dict={Z: Z_sample, Y:y_sample})
```

现在我们通过执行下面的步骤来根据标记的条件生成衣柜的图像。首先通过运行下面的脚本下载 Fashion-MNIST 数据集并保存在 data/fashion 目录下（见图 3-3）：

```
python download.py
```

```
a0999b1381a5:CGAN-code kuntalg$ python download.py
http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-images-idx3-uby
te.gz
Downloading  train-images-idx3-ubyte.gz
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
 82 25.2M   82 20.7M    0     0  1640k      0  0:00:15  0:00:12  0:00:03 1725k
```

图3-3：下载Fashion-MNIST数据集

接下来利用下面的命令来训练 CGAN 模型，在每 1000 次迭代后将会在 output 目录生成一批样本图像（见图 3-4）：

```
python simple-cgan.py
```

```
a0999b1381a5:CGAN-code kuntalg$ python simple-cgan.py
Extracting ./data/fashion/train-images-idx3-ubyte.gz
Extracting ./data/fashion/train-labels-idx1-ubyte.gz
Extracting ./data/fashion/t10k-images-idx3-ubyte.gz
Extracting ./data/fashion/t10k-labels-idx1-ubyte.gz
2017-08-30 11:50:51.450753: W tensorflow/core/platform/cpu_feature_guard.cc:45] The TensorFlow library wasn't compiled to
 use SSE4.2 instructions, but these are available on your machine and could speed up CPU computations.
2017-08-30 11:50:51.450788: W tensorflow/core/platform/cpu_feature_guard.cc:45] The TensorFlow library wasn't compiled to
 use AVX instructions, but these are available on your machine and could speed up CPU computations.
2017-08-30 11:50:51.450799: W tensorflow/core/platform/cpu_feature_guard.cc:45] The TensorFlow library wasn't compiled to
 use AVX2 instructions, but these are available on your machine and could speed up CPU computations.
2017-08-30 11:50:51.450808: W tensorflow/core/platform/cpu_feature_guard.cc:45] The TensorFlow library wasn't compiled to
 use FMA instructions, but these are available on your machine and could speed up CPU computations.
2017-08-30 11:50:52.412 Python[82549:2076967] ApplePersistenceIgnoreState: Existing state will not be touched. New state
will be written to /var/folders/4h/m2y74lpj7qs_ysw3gpfbcy70k2hbrw/T/org.python.python.savedState
Iter: 0
D loss: 1.382
G_loss: 0.1634
()
```

图3-4：CGAN训练

下面是通过 CGAN 将标签设为 4（**外套**）经过 8 **万**次迭代以及将标签设为 7（**运动鞋**）经过 6 **万**次迭代所生成的结果（见图 3-5）。

图3-5：CGAN生成图像

3.2.2 利用边界均衡固化 GAN 训练

GAN 在机器学习研究者中的人气越来越高。GAN 研究者主要分为两大类：一类将 GAN 应用到各种有挑战性的问题上，另一类试图固化 GAN 的训练过程。固化 GAN 的训练过程十分重要，原始的 GAN 模型具有下面几个问题。

- **模型坍塌：**生成器坍塌到一个极其狭小的分布内，生成的样本不再变化。这个问题显然违背了 GAN 的本质。
- **评估收敛指标：**现在没有一个很好的指标可以告诉我们生成器的损失和判别器的损失是否收敛。

沃瑟斯坦（Wasserstein）GAN（*arXiv: 1704.00028,2017*）是一个最新提出的 GAN 算法，其很有希望解决上面提到的问题。该算法试图通过给网络提供简单梯度（如果输出被认为是真则加 1，反之则减 1）的方法来最小化沃瑟斯坦距离（也被称为“推土机”距离）。

BEGAN（*arXiv: 1703.10717,2017*）背后的主要思想是通过一个**自动编码器**作为判别器来生成一个新的损失函数。真实的损失由沃瑟斯坦距离（用来解决模型坍塌问题）和重构真实图像与生成图像的损失衍生而来（见图 3-6）。

$$\mathcal{L}(v)=|v-D(v)|^{\eta} \quad \text{其中，} \begin{cases} D:\mathbb{R}^{N_x}\mapsto\mathbb{R}^{N_x} & \text{自动编码器函数} \\ \eta\in\{1,2\} & \text{目标规范} \\ v\in\mathbb{R}^{N_x} & N_x\text{维样板} \end{cases}$$

图3-6：沃瑟斯坦距离公式

一个超参数 gamma 通过影响参数 k 的权重来给用户提供控制模型多样性的能力（见图 3-7）。

$$\begin{cases} \mathcal{L}_D=\mathcal{L}(x)-k_t\mathcal{L}(G(z_D)) & \theta_D \\ \mathcal{L}_G=\mathcal{L}(G(z_G)) & \theta_G \\ k_{t+1}=k_t+\lambda_k(\gamma\mathcal{L}(x)-\mathcal{L}(G(z_G))) & \text{对每一次训练迭代}t \end{cases}$$

其中，gamma $\Rightarrow$ $\gamma=\dfrac{\mathbb{E}[\mathcal{L}(G(z))]}{\mathbb{E}[\mathcal{L}(x)]}$

图3-7：超参数gamma对模型的影响

和大多数 GAN 交替训练生成器和判别器不同，BEGAN 允许在每一步以对抗的方式同

时训练两个网络：

$$\arg\min_{\theta_D}\mathcal{L}_D+\arg\min_{\theta_G}\mathcal{L}_G$$

最终 M 可以用来衡量大致的收敛程度，帮助我们了解整个网络的性能：

$$\mathcal{M}_{\text{global}}=\mathcal{L}(x)+\left|\gamma\mathcal{L}(x)-\mathcal{L}\left(G\left(z_G\right)\right)\right|$$

3.3 BEGAN 的训练过程

下面是训练 BEGAN 的过程：

1. 判别器（自动编码器）更新权重使得真实图像的重构损失最小化，这样可以重构出更真实的图像。
2. 同时判别器开始最大化生成图像的重构损失。
3. 生成器网络以对抗的方式来最小化生成图像的重构损失。

3.3.1 BEGAN 的架构

如图 3-8 所示，判别器是一个拥有编码器和解码器的深度卷积网络。解码器拥有多个 3 × 3 大小的卷积层以及一个**指数线性单元**（ELU）。向下采样通过步幅为 2 的卷积完成。自动编码器的内嵌状态被映射到全连接层。生成器和解码器拥有相同的深度卷积架构，但是生成器权重上有区别，并且使用最邻近方式来实现向上采样。

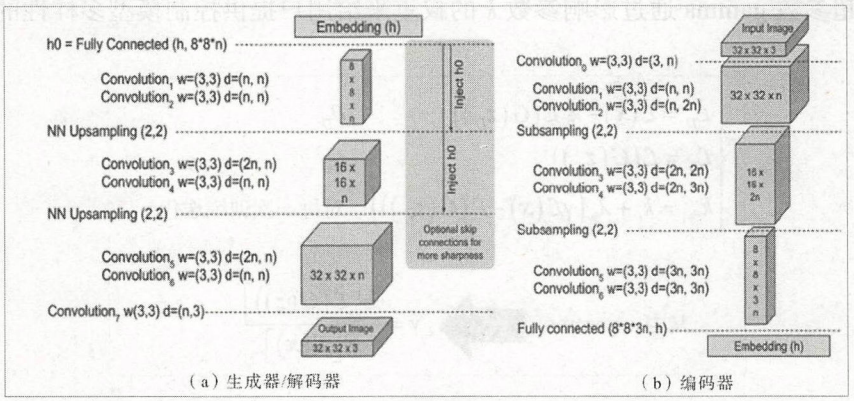


图3-8：BEGAN的架构

（来源：*arXiv: 1703.10717,2017/12/24*）

在图 3-8 中生成器和判别器的解码器都是通过左图展示的，判别器的编码器网络通过右图展示。

3.3.2 利用 TensorFlow 实现 BEGAN

下面让我们进入代码来实现之前所提到的概念，并且利用这种架构生成吸引人的逼真图像。

生成器网络除了最后一层外，均由使用 elu 激发函数 3×3 大小的卷积层，以及最邻近向上采样层构成。卷积层的数量需要根据图像的高度计算得出：

```
self.repeat_num = int(np.log2(height)) - 2.

def GeneratorCNN(z, hidden_num, output_num, repeat_num, data_format, reuse):
    with tf.variable_scope("G", reuse=reuse) as vs:
        num_output = int(np.prod([8, 8, hidden_num]))
        x = slim.fully_connected(z, num_output, activation_fn=None)
        x = reshape(x, 8, 8, hidden_num, data_format)

        for idx in range(repeat_num):
            x = slim.conv2d(x, hidden_num, 3, 1, activation_fn=tf.nn.elu,
data_format=data_format)
            x = slim.conv2d(x, hidden_num, 3, 1, activation_fn=tf.nn.elu,
data_format=data_format)
            if idx < repeat_num - 1:
                x = upscale(x, 2, data_format)

        out = slim.conv2d(x, 3, 3, 1, activation_fn=None,
data_format=data_format)

    variables = tf.contrib.framework.get_variables(vs)
    return out, variables
```

判别器网络的编码器除了最后一个卷积层外，均由 elu 激发函数的卷积层以及使用最大值池化的向下采样层构成：

```
def DiscriminatorCNN(x, input_channel, z_num, repeat_num, hidden_num,
data_format):
```

```
    with tf.variable_scope("D") as vs:
        # 编码器
        x = slim.conv2d(x, hidden_num, 3, 1, activation_fn=tf.nn.elu,
data_format=data_format)

        prev_channel_num = hidden_num
        for idx in range(repeat_num):
            channel_num = hidden_num * (idx + 1)
            x = slim.conv2d(x, channel_num, 3, 1, activation_fn=tf.nn.elu,
data_format=data_format)
            x = slim.conv2d(x, channel_num, 3, 1, activation_fn=tf.nn.elu,
data_format=data_format)
            if idx < repeat_num - 1:
                x = slim.conv2d(x, channel_num, 3, 2, activation_fn=tf.nn.elu,
data_format=data_format)
                #x = tf.contrib.layers.max_pool2d(x, [2, 2], [2, 2],
padding='VALID')

        x = tf.reshape(x, [-1, np.prod([8, 8, channel_num])])
        z = x = slim.fully_connected(x, z_num, activation_fn=None)
```

判别器网络的解码器和生成器网络类似，除了最后一个卷积层外，均由使用 `elu` 激发函数的卷积层以及最邻近向上采样层构成：

```
        num_output = int(np.prod([8, 8, hidden_num]))
        x = slim.fully_connected(x, num_output, activation_fn=None)
        x = reshape(x, 8, 8, hidden_num, data_format)

        for idx in range(repeat_num):
            x = slim.conv2d(x, hidden_num, 3, 1, activation_fn=tf.nn.elu,
data_format=data_format)
            x = slim.conv2d(x, hidden_num, 3, 1, activation_fn=tf.nn.elu,
data_format=data_format)
            if idx < repeat_num - 1:
                x = upscale(x, 2, data_format)

        out = slim.conv2d(x, input_channel, 3, 1, activation_fn=None,
data_format=data_format)
```

```
    variables = tf.contrib.framework.get_variables(vs)
```

接下来生成器的损失以及判别器区分真伪图像的损失均通过 Adam 优化器来进行优化，如下面的代码所示：

```
    d_out, self.D_z, self.D_var = DiscriminatorCNN(
                tf.concat([G, x], 0), self.channel, self.z_num,
self.repeat_num,
                self.conv_hidden_num, self.data_format)
            AE_G, AE_x = tf.split(d_out, 2)

            self.G = denorm_img(G, self.data_format)
            self.AE_G, self.AE_x = denorm_img(AE_G, self.data_format),
denorm_img(AE_x, self.data_format)

    if self.optimizer == 'adam':
                optimizer = tf.train.AdamOptimizer
            else:
                raise Exception("[!] Caution! Paper didn't use {} opimizer other
than Adam".format(config.optimizer))

            g_optimizer, d_optimizer = optimizer(self.g_lr),
optimizer(self.d_lr)

            self.d_loss_real = tf.reduce_mean(tf.abs(AE_x - x))
            self.d_loss_fake = tf.reduce_mean(tf.abs(AE_G - G))

            self.d_loss = self.d_loss_real - self.k_t * self.d_loss_fake
            self.g_loss = tf.reduce_mean(tf.abs(AE_G - G))
```

现在让我们运行下面的代码来生成名人的图像。

1. 首先克隆下面仓库的代码，并且进入代码目录：

```
git clone https://github.com/carpedm20/BEGAN-tensorflow.git
cd BEGAN-tensorflow
```

2. 接下来运行下面的脚本下载 CelebA 数据集到 data 目录，并将其切分成训练集、校验集以及测试集。

```
python download.py
```

3. 确认机器上已经安装了 p7zip。
4. 现在通过下面的命令开始训练，生成的样本将会被保存到 logs 目录下：

```
python main.py --dataset=CelebA --use_gpu=True
```

如果你遇到 **onv2DCustomBackpropInputOp only supports NHWC** 错误，请参考下面的链接：

https://github.com/carpedm20/BEGAN-tensorflow/issues/29

执行完上述操作，在训练进行的过程中你将会看到各种信息，包括模型（Model）目录、日志目录、各种损失等，如图 3-9 所示。

```
[ubuntu@ip-172-31-6-47:~/software/kuntalg/BEGAN-tensorflow$ python main.py --dataset=CelebA --use_g]
pu=True
2017-08-26 12:32:52.771534: W tensorflow/core/platform/cpu_feature_guard.cc:45] The TensorFlow lib
rary wasn't compiled to use SSE4.1 instructions, but these are available on your machine and could
 speed up CPU computations.
2017-08-26 12:32:52.771590: W tensorflow/core/platform/cpu_feature_guard.cc:45] The TensorFlow lib
rary wasn't compiled to use SSE4.2 instructions, but these are available on your machine and could
 speed up CPU computations.
2017-08-26 12:32:52.771600: W tensorflow/core/platform/cpu_feature_guard.cc:45] The TensorFlow lib
rary wasn't compiled to use AVX instructions, but these are available on your machine and could sp
eed up CPU computations.
[*] MODEL dir: logs/CelebA_0826_123249
[*] PARAM path: logs/CelebA_0826_123249/params.json
  0%|                                                                    | 0/500000 [00:00<?, ?it/s]
[0/500000] Loss_D: 0.460274 Loss_G: 0.052054 measure: 0.6384, k_t: 0.0002
[*] Samples saved: logs/CelebA_0826_123249/0_G.png
[*] Samples saved: logs/CelebA_0826_123249/0_D_real.png
[*] Samples saved: logs/CelebA_0826_123249/0_D_fake.png
  0%|                                                  | 50/500000 [06:34<1073:17:56,  7.73s/it]
[50/500000] Loss_D: 0.254775 Loss_G: 0.043506 measure: 0.3390, k_t: 0.0049
  0%|                                                  | 99/500000 [12:57<1084:16:39,  7.81s/it]
```

图3-9：启动模型训练

BEGAN 产生的图像如图 3-10 所示，它们在视觉上十分逼真，而且也很有吸引力。

图3-10：35万次迭代后的生成图像（64×64），gamma=0.5

下面这些样本是经过 25 万次迭代后生成的样本图像（128 × 128），如图 3-11 所示。

图3-11：25万次迭代后的生成图像（128×128），gamma=0.5

3.4 利用 CycleGAN 实现图像风格的转换

循环一致生成网络（Cycle Consistent Generative Network，CycleGAN）最初在论文 *Unpaired image-to-image translation using CycleGAN——arXiv: 1703.10593, 2017* 中提出。主要用来寻找不需要其他额外的信息就能将一张图像从源领域映射到目标领域的方法（例如将灰度图变成彩色图，图像语义标签的生成，边缘图生成照片，马变成斑马，等等）。

CycleGAN 背后的核心思想是两个转换器 F 和 G，其中 F 会将图像从域 A 转换到域 B，而 G 会将图像从域 B 转换到域 A。因此，对于一个在域 A 的图像 $\boldsymbol{x}$，我们期望函数 $G(F(\boldsymbol{x}))$ 的结果与 $\boldsymbol{x}$ 相同；同样，对于一个在域 B 的图像 $\boldsymbol{y}$，我们期望函数 $F(G(\boldsymbol{y}))$ 的结果和 $\boldsymbol{y}$ 相同。

3.4.1 CycleGAN 的模型公式

CycleGAN 模型的主要目标是通过训练样本$\{\boldsymbol{x}_i\}N_i=1 \in X$ 和$\{\boldsymbol{y}_j\}M_j=1 \in Y$ 来学习两个领域 X 和 Y 之间的映射关系。它也有两个对抗判别器 D_X和 D_Y：其中 D_X试图区分原始图像$\{\boldsymbol{x}\}$和转换后的图像$\{F(\boldsymbol{y})\}$，同时 D_Y试图区分$\{\boldsymbol{y}\}$和$\{G(\boldsymbol{x})\}$。

CycleGAN 模型有以下两个损失函数。

- **对抗损失**：它和生成图像的分布以及目标域的分布相匹配。

$$\mathcal{L}_{\text{GAN}}\left(G,D_Y,X,Y\right)=\mathbb{E}_{y\sim p_{\text{data}}(y)}\left[\log D_Y\left(y\right)\right]+\mathbb{E}_{x\sim p_{\text{data}}(x)}\left[\log\left(1-D_Y\left(G\left(x\right)\right)\right)\right]$$

- **循环一致损失**：它用来避免学习到的转换器 G 和 F 相互矛盾。

$$\mathcal{L}_{\text{cyc}}\left(G,F\right)=\mathbb{E}_{x\sim p_{\text{data}}(x)}\left[\left\|F\left(G\left(x\right)\right)-x\right\|1\right]+\mathbb{E}_{y\sim p_{\text{data}}(y)}\left[\left\|G\left(F\left(y\right)\right)-y\right\|1\right]$$

完整的 CycleGAN 目标函数如下：

$$G^*,F^*=\arg\min_{F,G}\max_{D_X,D_Y}\mathcal{L}\left(G,F,D_X,D_Y\right)$$

$$\Longrightarrow \mathcal{L}\left(G,F,D_X,D_Y\right)=\mathcal{L}_{\text{GAN}}\left(G,D_Y,X,Y\right)+\mathcal{L}_{\text{GAN}}\left(F,D_X,Y,X\right)+\lambda\mathcal{L}_{\text{cyc}}\left(G,F\right)$$

3.4.2 利用 TensorFlow 将苹果变成橘子

在本例中我们会将一个域 A 中的图像转换到另一个域 B 中，更具体地说就是，我们将应用 CycleGAN 进行苹果和橘子的相互转换，具体步骤如下。

1. 克隆下面仓库的代码并进入 `CycleGAN-tensorflow` 目录：

```
git clone https://github.com/xhujoy/CycleGAN-tensorflow
```

```
cd CycleGAN-tensorflow
```

2. 利用 download_dataset.sh 脚本下载 apple2orange ZIP 格式的数据集，并将其解压缩到 datasets 目录。

```
bash ./download_dataset.sh apple2orange
```

3. 接下来利用下载的 apple2orange 数据集训练 CycleGAN 模型。在训练过程中模型将会被保存到 checkpoint 目录，并可以通过 TensorBoard 来可视化观察训练过程中的日志（见图 3-12）。

```
python main.py --dataset_dir=apple2orange
```

```
[ubuntu@ip-172-31-6-47:~/software/kuntalg/CycleGAN-tensorflow$ python main.py --dataset_dir=apple2orange
 --phase=test --which_direction=AtoB
2017-08-27 10:26:41.727036: W tensorflow/core/platform/cpu_feature_guard.cc:45] The TensorFlow library
wasn't compiled to use SSE4.1 instructions, but these are available on your machine and could speed up
CPU computations.
2017-08-27 10:26:41.727083: W tensorflow/core/platform/cpu_feature_guard.cc:45] The TensorFlow library
wasn't compiled to use SSE4.2 instructions, but these are available on your machine and could speed up
CPU computations.
2017-08-27 10:26:41.727101: W tensorflow/core/platform/cpu_feature_guard.cc:45] The TensorFlow library
wasn't compiled to use AVX instructions, but these are available on your machine and could speed up CPU
 computations.
generatorA2B/g_e1_c/Conv/weights:0
generatorA2B/g_e1_bn/scale:0
generatorA2B/g_e1_bn/offset:0
generatorA2B/g_e2_c/Conv/weights:0
generatorA2B/g_e2_bn/scale:0
```

图3-12：启动模型训练

4. 运行下面的命令，然后通过浏览器 http://localhost:6006/可视化观察多个损失变化（生成器和判别器的损失，见图 3-13）：

```
tensorboard --logdir=./logs
```

5. 最后我们需要载入 checkpoint 目录下已经训练好的模型来进行图像风格的转换。根据传入 which_direction 参数的不同（AtoB 或者 BtoA），会选择转换方向是从苹果到橘子，还是从橘子到苹果：

```
python main.py --dataset_dir=apple2orange --phase=test --which_
direction=AtoB
```

下面是一些在测试阶段生成的图像样本（见图 3-14）。

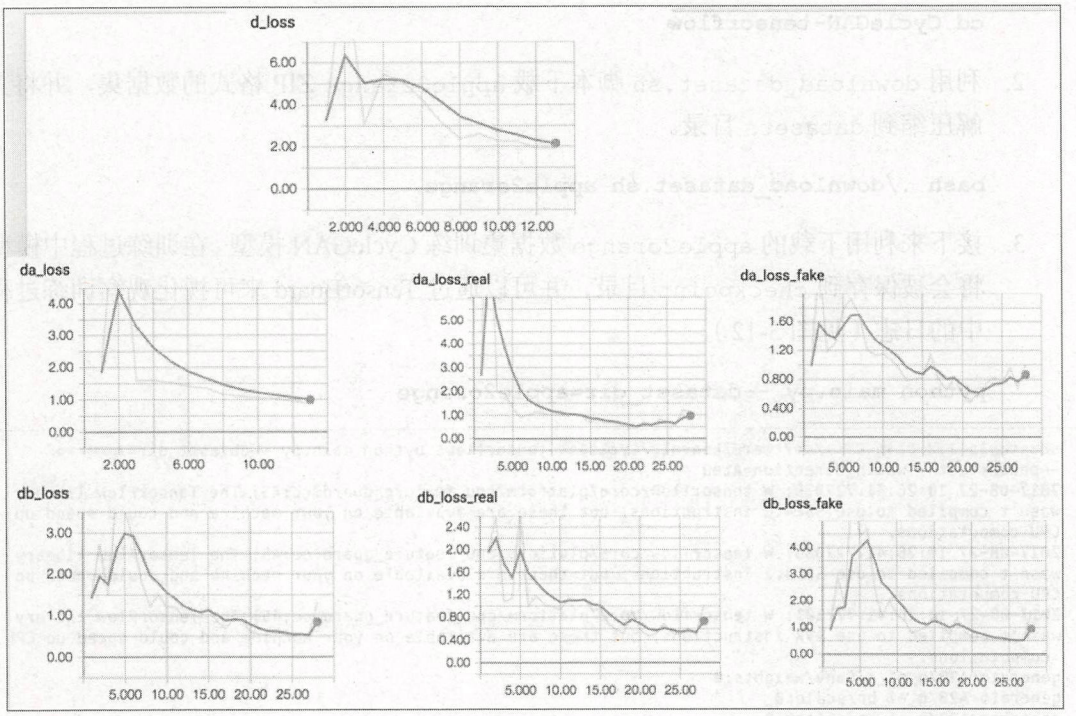


图3-13：模型损失变化

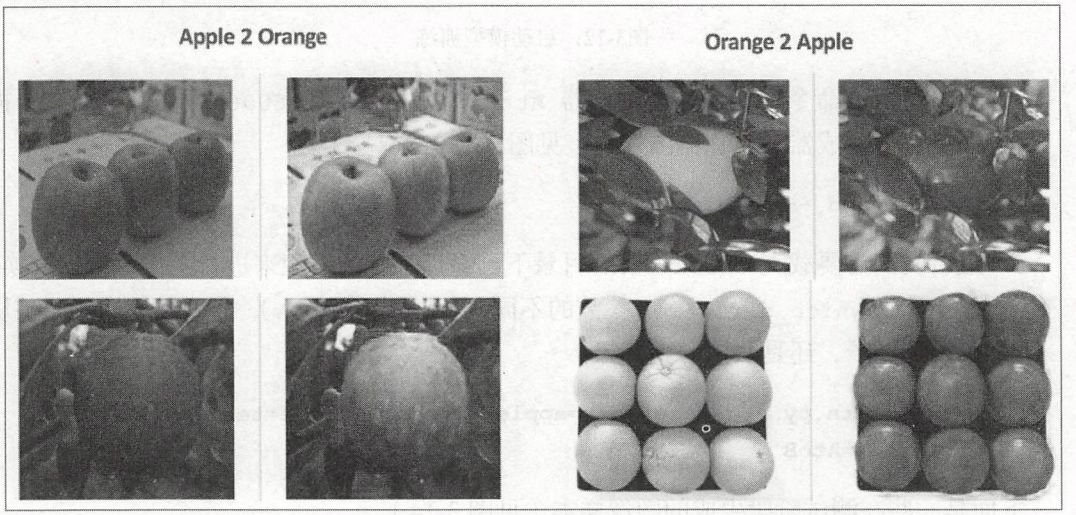


图3-14：左侧图像为传入AtoB参数从苹果向橘子转换的结果，
右侧图像为传入BtoA参数从橘子向苹果转换的结果

3.4.3 利用 CycleGAN 将马变为斑马

和上一个例子类似，在本节中我们将会利用 CycleGAN 进行马和斑马之间的转换，具体步骤如下。

1. 克隆下面仓库的代码并进入 `CycleGAN-tensorflow` 目录（如果前面已经做过，则可以忽略本步骤）：

```
git clone https://github.com/xhujoy/CycleGAN-tensorflow
cd CycleGAN-tensorflow
```

2. 利用 `download_dataset.sh` 脚本从 Berkley 下载 `horse2zebra` 数据集的 ZIP 文件，解压缩并保存到 `datasets` 目录下：

```
bash ./download_dataset.sh horse2zebra
```

3. 接下来我们利用 `horse2zebra` 数据集进行 CycleGAN 模型的训练，并利用 TensorBoard 来可视化训练过程中的损失变化（见图 3-15）：

```
python main.py --dataset_dir=horse2zebra
```

```
ubuntu@ip-172-31-6-47:~/software/kuntalg/CycleGAN-tensorflow$ python main.py --dataset_dir=horse2zebra
2017-08-27 06:12:07.756238: W tensorflow/core/platform/cpu_feature_guard.cc:45] The TensorFlow library
wasn't compiled to use SSE4.1 instructions, but these are available on your machine and could speed up
CPU computations.
2017-08-27 06:12:07.756280: W tensorflow/core/platform/cpu_feature_guard.cc:45] The TensorFlow library
wasn't compiled to use SSE4.2 instructions, but these are available on your machine and could speed up
CPU computations.
2017-08-27 06:12:07.756299: W tensorflow/core/platform/cpu_feature_guard.cc:45] The TensorFlow library
wasn't compiled to use AVX instructions, but these are available on your machine and could speed up CPU
 computations.
generatorA2B/g_e1_c/Conv/weights:0
generatorA2B/g_e1_bn/scale:0
generatorA2B/g_e1_bn/offset:0
generatorA2B/g_e2_c/Conv/weights:0
generatorA2B/g_e2_bn/scale:0
generatorA2B/g_e2_bn/offset:0
generatorA2B/g_e3_c/Conv/weights:0
```

图3-15：启动模型训练

4. 运行下面的命令，然后通过浏览器 `http://localhost:6006/`可视化观察生成器和判别器的损失（见图 3-16）：

```
tensorboard --logdir=./logs
```

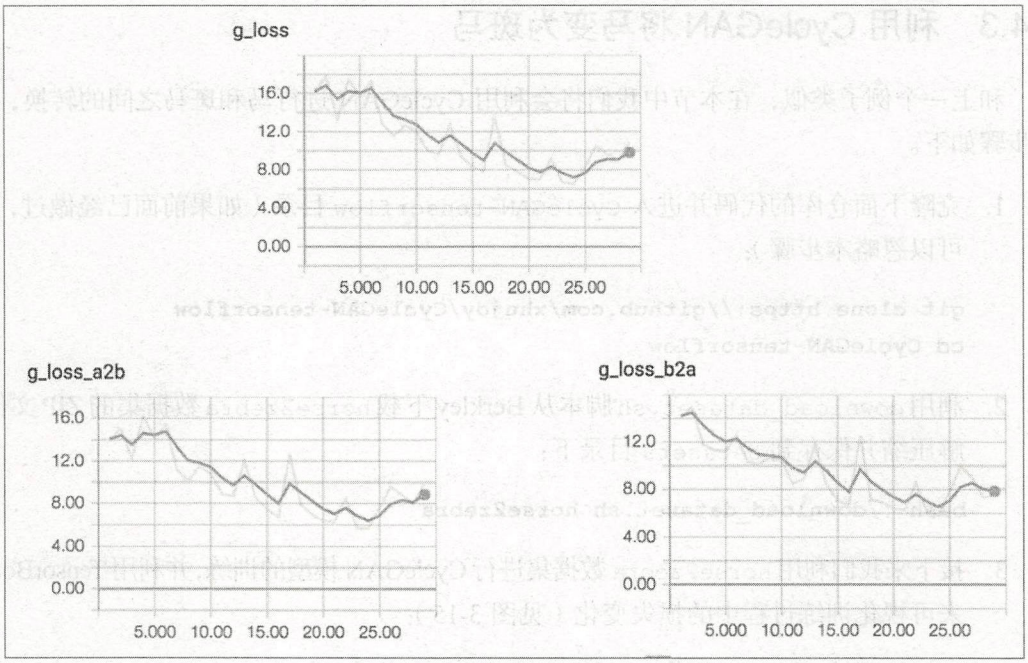


图3-16：模型损失变化

5. 最后我们需要载入 `checkpoint` 目录下已经训练好的模型来进行图像风格的转换。根据传入 `which_direction` 参数的不同（`AtoB` 或者 `BtoA`），会选择转换方向是从马到斑马，还是从斑马到马：

```
python main.py --dataset_dir=horse2zebra --phase=test --which_
direction=AtoB
```

下面是一些在测试（`test`）阶段生成的图像样本（见图 3-17）。

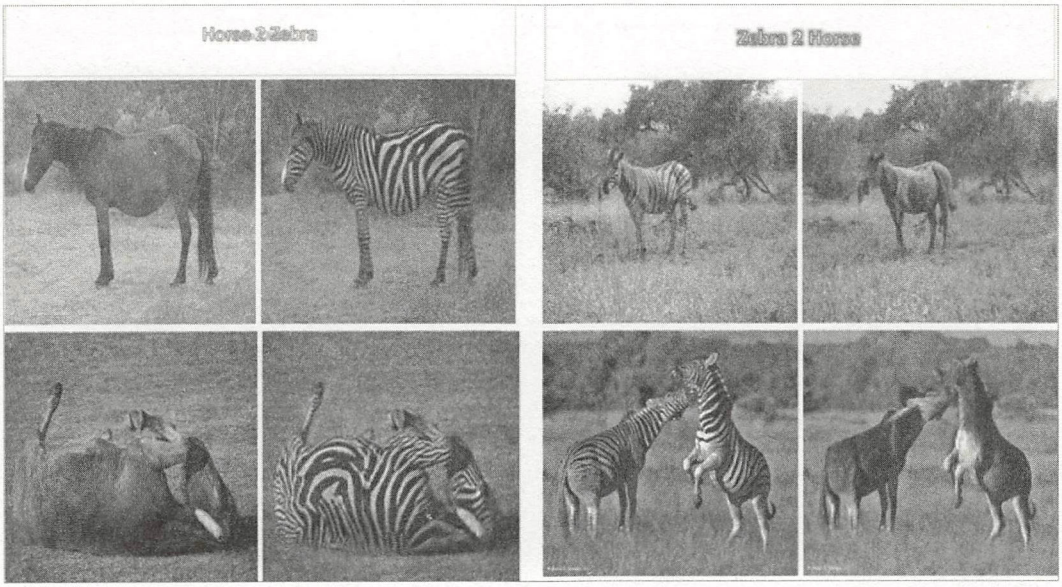


图3-17：左侧展示从马到斑马的转换，右侧展示从斑马到马的转换

3.5 总结

到这里你已经掌握了根据特定条件和特点，将条件向量分别传入生成器和判别器来进行图像生成的方法。同时你也了解了如何通过 BEGAN 来固化网络训练以避免模型坍塌问题。最后我们通过 CycleGAN 实现了苹果和橘子以及马和斑马之间图像风格的相互转换。在第 4 章中我们将通过堆叠或者组合多个 GAN 模型来解决一些日常生活中更加复杂的问题，例如从文本生成图像以及跨域关系的探索。

4

从文本构建逼真的图像

对于日常生活的一些复杂问题，仅使用一个生成对抗网络并不能很好地解决问题。一种更好的方式是将复杂问题拆分成多个子问题，并且在每个子问题上单独运行 GAN 进行学习。最终我们可能会堆叠或者组合多个 GAN 得到一个解决方案。

在本章中，我们首先学习利用层叠多个生成网络来从文本信息生成逼真图像的技术。接下来我们会组合两个生成网络来自动学习多个不同领域之间的联系（例如鞋和包之间的关系，以及男演员与女演员之间的关系）。

本章将会包含以下内容：

- 什么是 StackGAN，它的概念和架构。
- 利用 TensorFlow 通过文本生成逼真的图像。
- 利用 DiscoGAN 发现不同域之间的联系。
- 利用 PyTorch 从边缘生成手提包。
- 利用 `facescrub` 数据集实现演员性别的转换。

4.1 StackGAN 介绍

StackGAN 的思想最早自 Han Zhang、Tao Xu、Hongsheng Li、Shaoting Zhang、Xiaolei Huang、Xiaogang Wang 和 Dimitris Metaxas 的论文 *Text to Photo-realistic Image Synthesis with Stacked Generative Adversarial Networks* [*arXiv: 1612.03242,2017*]引入，其中，GAN 被用于通过文本描述来生成图像。

通过文本制作逼真的图像在计算机视觉领域是一个极具挑战的问题，并且有着极其广

泛的应用场景。通过文本生成图像的问题可以分解为两个更易于控制的子问题，进而可以使用 StackGAN。通过这种方法，我们根据特定条件（文本描述和上一阶段的输出）堆叠出了一个两阶段生成网络来解决从文本生成逼真图像这个问题。

在开始进入模型架构和实现细节之前，我们先来定义一些概念和术语。

- I_0：原始图像。
- t：文本描述。
- t：文本嵌入。
- $\mu(t)$：文本嵌入的平均值。
- $\Sigma(t)$：文本嵌入的对角协方差矩阵。
- p_{data}：真实数据分布。
- p_z：高斯分布噪声。
- z：高斯分布中的随机噪声采样。

4.1.1 条件强化

我们从第 2 章中已经了解到条件 GAN 需要给生成器网络和判别器网络一个额外的条件变量 c 以产生 $G(z;c)$和 $D(x;c)$。这个公式帮助生成器根据变量 c 来生成图像。条件强化会根据少量的图像-文本对产生更多的训练对，这对于根据文本生成图像的训练十分有帮助，因为通常情况下相同的语句可能会用来描述不同的外貌。文本描述首先通过一个编码器进行编码，然后利用 char-CNN-RNN 模型进行非线性转换生成文本嵌入 t，并且产生潜在的条件变量作为 Stage-I 生成器网络的输入。

由于文本嵌入的潜在空间通常都是高维度的，因此，为了解决在有限数据情况下潜在数据不连续的问题，需要应用条件强化的技术通过高斯分布 $N(\mu(t),\Sigma(t))$生成条件变量更多的样本 $\hat{c}$。

1. Stage-I

在本阶段中，GAN 网络需要学习以下内容：

- 根据文本描述的条件生成物体大致的形状和基本的颜色。
- 通过先前的分布和随机噪声样本生成背景区域。

在这个阶段生成的低清晰度的粗粒度图像看起来并不真实，有可能物体形状被扭曲，

或者丢失了某些部分。

Stage-I GAN 分别通过下面的公式来训练判别器 D_0（最大化损失）和生成器 G_0（最小化损失）：

Discriminator Loss

$$\mathcal{L}_{D_0} = \mathbb{E}_{(I_0,t)\sim p_{\text{data}}}\left[\log D_0\left(I_0,\varphi_t\right)\right] + \mathbb{E}_{z\sim p_z,t\sim p_{\text{data}}}\left[\log\left(1-D_0\left(G_0\left(z,\hat{c}_0\right),\varphi_t\right)\right)\right]$$

Generator Loss

$$\mathcal{L}_{G_0} = \mathbb{E}_{z\sim p_z,t\sim p_{\text{data}}}\left[\log\left(1-D_0\left(G_0\left(z,\hat{c}_0\right),\varphi_t\right)\right)\right] + \lambda D_{KL}\left(N\left(\mu_0\left(\varphi_t\right),\Sigma_0\left(\varphi_t\right)\right)\middle\|N\left(0,1\right)\right)$$

利用 Kullback-Leibler 散度作为标准分布和条件高斯分布的正则化因子

2. Stage-II

在本阶段中，GAN 网络只关注如何勾勒细节，以及矫正 Stage-I 产生的低清晰度图像的问题（缺少某些部分、形状扭曲以及被忽略的文本细节），以便生成一张符合文本描述信息的高清晰度逼真图像。

Stage-II GAN 使用 $G_0(z;\hat{c}_0)$的低清晰度结果以及高斯潜在变量 $\hat{c}$，通过下面的公式来训练判别器 D（最大化损失）和生成器 G（最小化损失）：

Discriminator Loss

$$\mathcal{L}_D = \mathbb{E}_{(I,t)\sim p_{\text{data}}}\left[\log D\left(I,\varphi_t\right)\right] + \mathbb{E}_{s_0\sim p_{G_0},t\sim p_{\text{data}}}\left[\log\left(1-D\left(G\left(s_0,\hat{c}\right),\varphi_t\right)\right)\right]$$

Generator Loss

$$\mathcal{L}_G = \mathbb{E}_{s_0\sim p_{G_0},t\sim p_{\text{data}}}\left[\log\left(1-D\left(G\left(s_0,\hat{c}\right),\varphi_t\right)\right)\right] + \lambda D_{KL}\left(N\left(\mu\left(\varphi_t\right),\Sigma\left(\varphi_t\right)\right)\middle\|N\left(0,1\right)\right)$$

在 Stage-II 中高斯条件变量 $\hat{c}$ 取代了随机噪声 z。同时 Stage-II 中的条件增强有着不同的全连接层，生成的文本嵌入平均值和标准差也不同。

4.1.2 StackGAN 的架构细节

如图 4-1 所示，对于 Stage-I 的生成器网络 G_0，文本嵌入 $\boldsymbol{t}$ 首先被传入一个全连接层来生成高斯分布 $N(\mu_0(\boldsymbol{t}); \Sigma_0(\boldsymbol{t}))$所需要的 μ_0 和 σ_0（σ_0 是 Σ_0 的对角值）。接下来从高斯分布中进行采样，生成文本条件变量 $\hat{c}_0$。

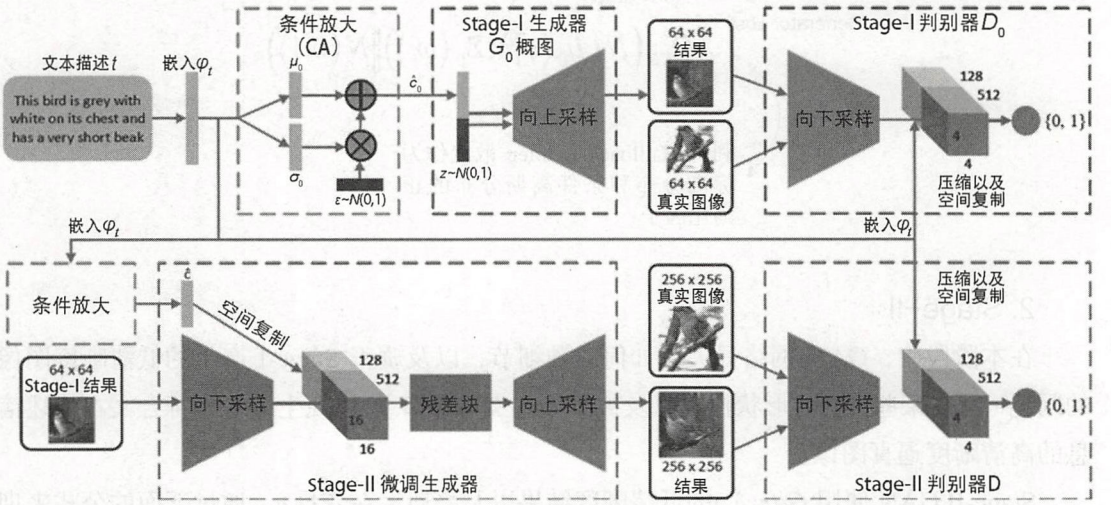


图4-1：StackGAN架构

（来源：*arXiv: 1612.03242,2017*）

对于 Stage-I 的判别器网络 D_0，文本嵌入 $\boldsymbol{t}$ 首先被压缩至 N_d 维度的全连接，接下来被空间复制成 $M_d \times M_d \times N_d$ 维度的张量（tensor）。图像通过一系列的向下采样处理被压缩入 $M_d \times M_d$ 维度空间，接下来与一个文本张量和一个过滤图串联。最终的张量通过一个 1×1 的卷积层联合从图像和文本中学习到的特征，使用一个全连接节点生成最终的决策分数。

Stage-II 生成器被设计成一个编码器-解码器网络，利用残差处理和文本嵌入 $\boldsymbol{t}$ 生成 N_g 维文本条件变量 $\hat{c}$，并空间复制成 $M_d \times M_d \times N_d$ 维度的张量。Stage-I 的结果 s_0 经过多个向下采样处理（编码器），直到被压缩至 $M_g \times M_g$ 维度的空间。图像特征和文本特征在维度通道内进行串联，并传入多个残差层来学习与图像及特征相关的多模式表征。最终，结果张量通过一系列向上采样层（解码器）生成 $W \times H$ 高清晰度图像。

Stage-II 的判别器和 Stage-I 类似，只做额外的向下采样处理来应对本阶段的大规模图像。在判别器的训练过程中正样本由与文本描述信息一致的真实图像组成；而负样本可以分两类：一类是真实图像和不匹配的文本嵌入，另一类是生成的图像和相关的文本嵌入。

Stage-I 生成器首先根据文本生成一张只包含大概形状和基本颜色的物体图像，并根据一个随机噪声向量填充背景。之后，Stage-II 生成器纠正错误并在 Stage-I 的基础上加入更多的细节，根据 Stage-I 的结果生成更逼真、更清晰的图像。

向上采样部分包含了最邻近向上采样以及步长为 1 的 33 个卷积层。除了最后一层，每一层都应用了批规范化和 ReLU 激发函数。残差处理部分同样包含了 33 个步长为 1 并应用批规范化和 ReLU 激发函数的卷积层。向下采样部分包含了 44 个步长为 2，并应用了批规范化以及 Leaky ReLU 激发函数的卷积层，在第一个卷积层中并没有使用批规范化。

4.1.3 利用 TensorFlow 从文本生成图像

下面让我们通过代码来实现从文本生成图像这项令人兴奋的任务：

1. 克隆代码仓库 https://github.com/Kuntal-G/StackGAN 并进入 StackGAN 目录。

```
git clone https://github.com/Kuntal-G/StackGAN.git
cd StackGAN
```

当前版本的代码和较老版本的 TensorFlow（0.11）并不兼容，你需要 1.0 版本以下的 TensorFlow 才能成功运行代码。可以根据下面的命令来更新你的 TensorFlow：

```
sudo pip install tensorflow==0.12.0
```

同时请确认你的系统已经安装了 torch。关于 torch 的更多信息请参考 http://torch.ch/docs/getting-started.html。

2. 通过 pip 命令安装下列软件包。

```
sudo pip install prettytensor progressbar python-dateutil easydict
pandas torchfile requests
```

3. 通过下面的命令从 https://drive.google.com/file/d/0B3y_msrWZaXLT1BZd-VdycDY5TEE/viewm 下载经过预处理的 char-CNN-RNN 文本嵌入鸟类模型。

```
python google-drive-download.py 0B3y_msrWZaXLT1BZdVdycDY5TEE Data/
birds.zip
```

4. 利用 unzip 命令解压缩下载的文件。

```
unzip Data/birds.zip
```

5. 从 Caltech-UCSD 下载并解压缩鸟类图像数据。

```
wget http://www.vision.caltech.edu/visipedia-data/CUB-200-2011/
CUB_200_2011.tgz -O Data/birds/CUB_200_2011.tgz
tar -xzf CUB_200_2011.tgz
```

6. 现在我们对图像数据进行预处理，划分训练集和测试集，并将图像以二进制形式存放（见图 4-2）。

```
python misc/preprocess_birds.py
```

```
[ubuntu@ip-172-31-1-246:~/software/kuntalg/StackGAN$ python misc/preprocess_birds.py
Total filenames:  11788 001.Black_footed_Albatross/Black_Footed_Albatross_0046_18.jpg
Load filenames from: Data/birds/train/filenames.pickle (8855)
Load 100......
Load 200......
Load 300......
Load 400......
Load 500......
Load 600......
Load 700......
Load 800......
Load 900......
Load 1000......

Load 8300......
Load 8400......
Load 8500......
Load 8600......
Load 8700......
Load 8800......
images 8855 (304, 304, 3) (76, 76, 3)
save to:  Data/birds/train/304images.pickle
save to:  Data/birds/train/76images.pickle
Load filenames from: Data/birds/test/filenames.pickle (2933)
Load 100......
Load 200......
Load 300......
Load 400......
Load 500......
Load 600......

Load 2800......
Load 2900......
images 2933 (304, 304, 3) (76, 76, 3)
save to:  Data/birds/test/304images.pickle
save to:  Data/birds/test/76images.pickle
```

图4-2：对图像数据进行预处理

7. 通过下面的命令从 https://drive.google.com/file/d/0B3y_msrWZaXLNUN-Ka3BaRjAyTzQ/view 下载经过预训练的 char-CNN-RNN 文本嵌入模型，并保存至 models/目录。

```
python google-drive-download.py 0B3y_msrWZaXLNUNKa3BaRjAyTzQ
models/ birds_model_164000.ckpt
```

8. 从 https://drive.google.com/file/d/0B0ywwgffWnLLU0F3UHA3NzFTNEE/view 下载 char-CNN-RNN 鸟类文本编码器并保存至 models/text_encoder 目录。

```
python google-drive-download.py 0B0ywwgffWnLLU0F3UHA3NzFTNEE
models/text_encoder/  lm_sje_nc4_cub_hybrid_gru18_a1_
c512_0.00070_1_10_trainvalids.txt_iter30000.t7
```

9. 在 example_captions.txt 文件中加入一些描述语句来生成鸟类的图像。

```
A white bird with a black crown and red beak
this bird has red breast and yellow belly
```

10. 最后我们执行 demo 目录下的 birds_demo.sh 文件，以通过在 example_captions.txt 文件里的文本描述信息生成逼真的鸟类图像（见图 4-3）。

```
sh demo/birds_demo.sh
```

```
[ubuntu@ip-172-31-6-47:~/software/kuntalg/StackGAN$ sh demo/birds_demo.sh
{
  doc_length : 201
  filenames : "Data/birds/example_captions.t7"
  queries : "Data/birds/example_captions.txt"
  net_txt : "models/text_encoder/lm_sje_nc4_cub_hybrid_gru18_a1_c512_0.00070_1_10_trainvalids.txt_iter3
0000.t7"
}
```

图4-3：生成图像

11. 现在生成的图像会被保存到 Data/birds/example_captions/目录下，如图 4-4 所示。

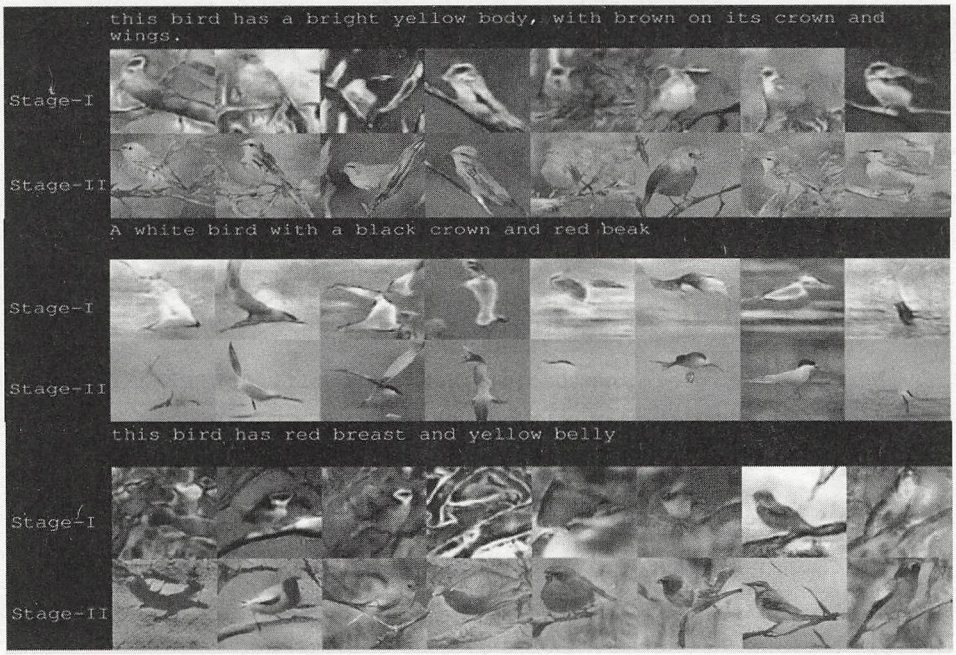


图4-4：生成图像样例

你现在可以通过文本描述来生成逼真的鸟类图像了。尝试写下你自己对鸟类的描述，看一下会生成怎样的结果。

4.2 利用 DiscoGAN 探索跨域的关系

发现跨域之间的关系对人类来说是一件很自然的事情，我们可以不需要经过监督学习的过程就很轻易地发现两个不同域数据之间的联系（例如分辨出一个英文句子和翻译后的西班牙句子之间的关系，或者选择一个和裙子风格搭配的鞋子）。但是对于机器来说，自动化学习这个过程是十分有挑战性的，需要了解大量的正确基准以及描述关系的信息。

探索生成对抗网络（Discovery Generative Adversarial Network，DiscoGAN）*arXiv: 1703.05192, 2017* 探索了两个不同视觉域的关系，成功地将一个域的图像转化到另一个域，并且不需要任何两个域之间关系的信息。DiscoGAN 寻找两个 GAN 的组合，它们相互映射到对方的域。DiscoGAN 的基本思想是确保所有在域 1 内的图像都可以用域 2 里的图像进行表示，利用重构损失来衡量原始图像经过两次转换，即从域 1 转换到域 2 再转换回域 1 后被重构的效果。

4.2.1 DiscoGAN 架构以及模型公式

在进入模型公式细节和多种 DiscoGAN 相关损失（loss）函数之前，我们先来定义一些相关的概念和术语（见图 4-5）。

- G_{AB}：将域 A 的图像 x_A 转换成域 B 的图像 x_{AB} 的生成器函数。
- G_{BA}：将域 B 的图像 x_B 转换成域 A 的图像 x_{BA} 的生成器函数。
- $G_{AB}(x_A)$：包含了在域 A 的 x_A 经过转换后属于域 B 的所有可能结果。
- $G_{AB}(x_B)$：包含了在域 B 的 x_B 经过转换后属于域 A 的所有可能结果。
- D_A：域 A 内的判别器（discriminator）函数。
- D_B：域 B 内的判别器（discriminator）函数。

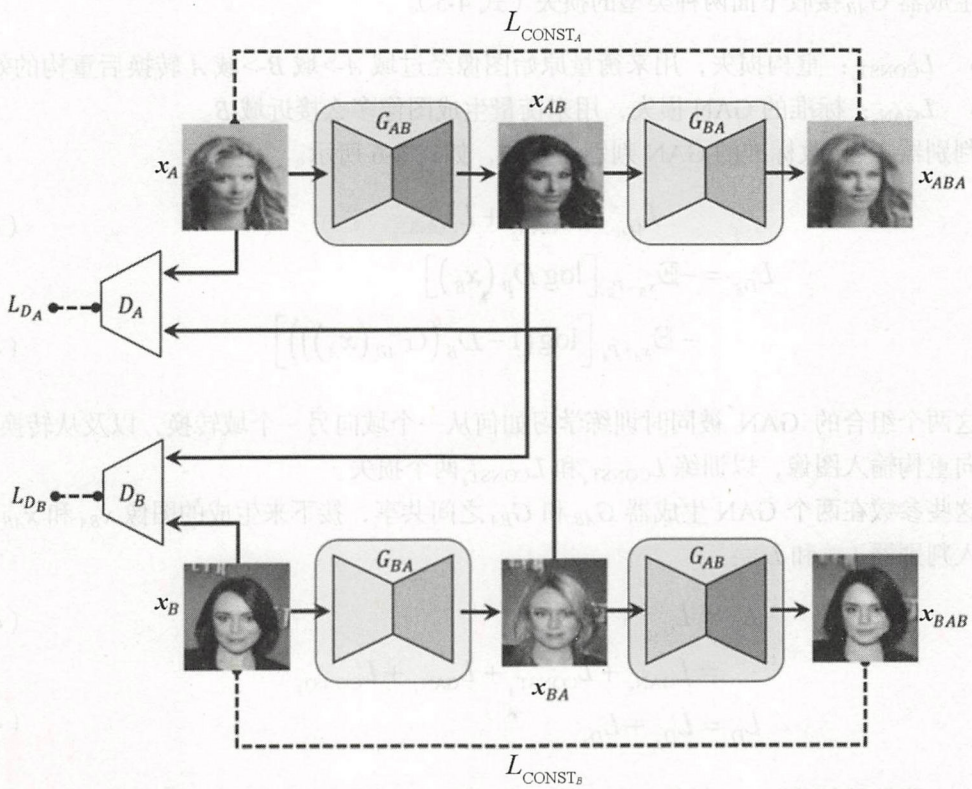


图4-5：由两个GAN模型组成的DiscoGAN架构

（来源：*arXiv: 1703.05192,2017*）

DiscoGAN 生成器模型包含了一个编码器-解码器对，以对图像进行双向转换。生成器

G_{AB}首先将域A内的图像$\boldsymbol{x}_A$转换为域B内的图像$\boldsymbol{x}_{AB}$。接下来生成的图像被转回域A的$\boldsymbol{x}_{ABA}$，我们利用一些距离指标例如 MSE 和余弦距离来计算转换后的图像和原图像的重构损失（式 4-3）。最后我们将生成器生成的图像 $\boldsymbol{x}_{AB}$ 传入判别器，并且得到和域 B 真实图像进行比较后的分数：

$$\boldsymbol{x}_{AB} = G_{AB}(\boldsymbol{x}_A) \tag{4-1}$$

$$\boldsymbol{x}_{ABA} = G_{BA}(\boldsymbol{x}_{AB}) = G_{BA} \circ G_{AB}(\boldsymbol{x}_A) \tag{4-2}$$

$$L_{\text{CONST}_A = d(G_{BA} \circ G_{AB}(\boldsymbol{x}_A), \boldsymbol{x}_A)} \tag{4-3}$$

$$L_{\text{GAN}_B} = -\mathbb{E}_{\boldsymbol{x}_A \sim P_A}\left[\log D_B\left(G_{AB}(\boldsymbol{x}_A)\right)\right] \tag{4-4}$$

生成器 G_{AB} 接收下面两种类型的损失（式 4-5）。

- L_{CONST_A}：重构损失，用来衡量原始图像经过域 A->域 B->域 A 转换后重构的效果。
- L_{GAN_B}：标准的 GAN 损失，用来衡量生成图像多么接近域 B。

判别器 D_B 接收标准的 GAN 判别器损失，如式 4-6 所示。

$$L_{G_{AB}} = L_{\text{GAN}_B} + L_{\text{CONST}_B} \tag{4-5}$$

$$L_{D_B} = -\mathbb{E}_{\boldsymbol{x}_B \sim P_B}\left[\log D_B(\boldsymbol{x}_B)\right] - \mathbb{E}_{\boldsymbol{x}_A \sim P_A}\left[\log\left(1 - D_B\left(G_{AB}(\boldsymbol{x}_A)\right)\right)\right] \tag{4-6}$$

这两个组合的 GAN 被同时训练学习如何从一个域向另一个域转换，以及从转换的结果逆向重构输入图像，以训练 L_{CONST_A} 和 L_{CONST_B} 两个损失。

这些参数在两个 GAN 生成器 G_{AB} 和 G_{BA} 之间共享，接下来生成的图像 $\boldsymbol{x}_{BA}$ 和 $\boldsymbol{x}_{AB}$ 被分别传入判别器 L_{DA} 和 L_{DB}：

$$L_G = L_{G_{AB}} + L_{G_{BA}} = L_{\text{GAN}_B} + L_{\text{CONST}_A} + L_{\text{GAN}_A} + L_{\text{CONST}_B} \tag{4-7}$$

$$L_D = L_{D_A} + L_{D_B} \tag{4-8}$$

生成器的总损失 L_G，由两个模型的 GAN 损失和重构损失分别加和组成（如式 4-7 所示）。判别器的总损失 L_D，由分别在域 A 和域 B 进行判别的判别器损失 L_{DA} 和 L_{DB} 加和组成（如式 4-8 所示）。为了达到一对一的双射关系，DiscoGAN 模型利用两个 L_{GAN} 损失和两

个 L_{CONST} 重构损失做限制。

单射意味着域 A 中的每一个成员都在域 B 中有着唯一的匹配成员，满射意味着 B 域中的每一个成员都在域 A 中至少有一个映射。

双射意味着将单射和满射结合起来，两个域之间的成员有着恰好一对一的映射关系（见图 4-6）。

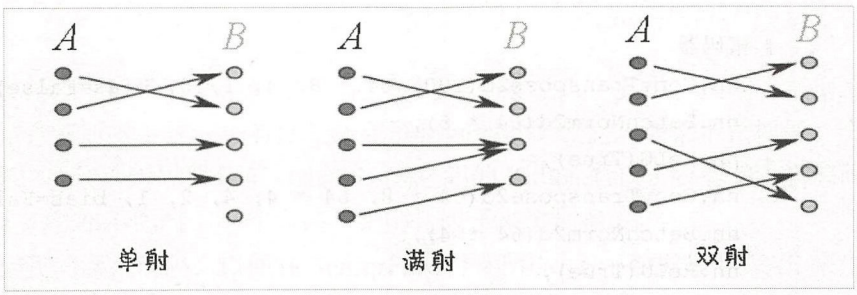


图4-6：单射、满射和双射

4.2.2 DiscoGAN 的实现

下面让我们进入代码来了解 DiscoGAN 的架构和相关概念（损失及衡量标准）。

生成器接收 64 × 64 × 3 大小的图像并传入一个编码器-解码器对。生成器的编码器部分由 5 个 4 × 4 大小过滤器的卷积层、每一层紧跟批规范化和 Leaky ReLU 组成。解码器部分由 5 个 4 × 4 大小过滤器的卷积层、每一层紧跟批规范化和 Leaky ReLU 组成，并输出目标域 64 × 64 × 3 大小的图像。下面是生成器的代码片段：

```
class Generator(nn.Module):

        self.main = nn.Sequential(
        # 编码器
            nn.Conv2d(3, 64, 4, 2, 1, bias=False),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Conv2d(64, 64 * 2, 4, 2, 1, bias=False),
            nn.BatchNorm2d(64 * 2),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Conv2d(64 * 2, 64 * 4, 4, 2, 1, bias=False),
            nn.BatchNorm2d(64 * 4),
            nn.LeakyReLU(0.2, inplace=True),
```

```
            nn.Conv2d(64 * 4, 64 * 8, 4, 2, 1, bias=False),
            nn.BatchNorm2d(64 * 8),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Conv2d(64 * 8, 100, 4, 1, 0, bias=False),
            nn.BatchNorm2d(100),
            nn.LeakyReLU(0.2, inplace=True),

        # 解码器
            nn.ConvTranspose2d(100, 64 * 8, 4, 1, 0, bias=False),
            nn.BatchNorm2d(64 * 8),
            nn.ReLU(True),
            nn.ConvTranspose2d(64 * 8, 64 * 4, 4, 2, 1, bias=False),
            nn.BatchNorm2d(64 * 4),
            nn.ReLU(True),
            nn.ConvTranspose2d(64 * 4, 64 * 2, 4, 2, 1, bias=False),
            nn.BatchNorm2d(64 * 2),
            nn.ReLU(True),
            nn.ConvTranspose2d(64 * 2, 64, 4, 2, 1, bias=False),
            nn.BatchNorm2d(64),
            nn.ReLU(True),
            nn.ConvTranspose2d(64,3, 4, 2, 1, bias=False),
            nn.Sigmoid()

            . . .

        )
```

判别器的结构和生成器的编码器部分类似，其由 5 个 4×4 大小过滤器的卷积层、每一层紧跟批规范化和 Leaky ReLU 组成。最终我们在最后的卷积层应用 sigmoid 函数来产生[0,1]之间的一个概率标量以判断数据的真伪。下面是判别器的代码片段：

```
class Discriminator(nn.Module):

        self.conv1 = nn.Conv2d(3, 64, 4, 2, 1, bias=False)
        self.relu1 = nn.LeakyReLU(0.2, inplace=True)

        self.conv2 = nn.Conv2d(64, 64 * 2, 4, 2, 1, bias=False)
```

```
        self.bn2 = nn.BatchNorm2d(64 * 2)
        self.relu2 = nn.LeakyReLU(0.2, inplace=True)

        self.conv3 = nn.Conv2d(64 * 2, 64 * 4, 4, 2, 1, bias=False)
        self.bn3 = nn.BatchNorm2d(64 * 4)
        self.relu3 = nn.LeakyReLU(0.2, inplace=True)

        self.conv4 = nn.Conv2d(64 * 4, 64 * 8, 4, 2, 1, bias=False)
        self.bn4 = nn.BatchNorm2d(64 * 8)
        self.relu4 = nn.LeakyReLU(0.2, inplace=True)

        self.conv5 = nn.Conv2d(64 * 8, 1, 4, 1, 0, bias=False)

        . . .

    return torch.sigmoid( conv5 ), [relu2, relu3, relu4]
```

接下来我们使用均方误差和二元交叉熵来定义生成器和重构的损失：

```
recon_criterion = nn.MSELoss()
gan_criterion = nn.BCELoss()


optim_gen = optim.Adam( gen_params, lr=args.learning_rate,
betas=(0.5,0.999), weight_decay=0.00001)
optim_dis = optim.Adam( dis_params, lr=args.learning_rate,
betas=(0.5,0.999), weight_decay=0.00001)
```

现在我们开始进行图像跨域转换，并通过计算重构损失来了解经过两次转换（ABA 或者 BAB）后图像的质量：

```
AB = generator_B(A)
BA = generator_A(B)

ABA = generator_A(AB)
BAB = generator_B(BA)

# 重构损失
```

```
    recon_loss_A = recon_criterion( ABA, A )
    recon_loss_B = recon_criterion( BAB, B )
```

随后，我们介绍每个域的生成器损失和判别器损失：

```
    # Real/Fake GAN Loss (A)
    A_dis_real, A_feats_real = discriminator_A( A )
    A_dis_fake, A_feats_fake = discriminator_A( BA )

    dis_loss_A, gen_loss_A = get_gan_loss( A_dis_real, A_dis_fake, gan_criterion,
cuda )
    fm_loss_A = get_fm_loss(A_feats_real, A_feats_fake, feat_criterion)

    # Real/Fake GAN Loss (B)
    B_dis_real, B_feats_real = discriminator_B( B )
    B_dis_fake, B_feats_fake = discriminator_B( AB )

    dis_loss_B, gen_loss_B = get_gan_loss( B_dis_real, B_dis_fake, gan_criterion,
cuda )
    fm_loss_B = get_fm_loss( B_feats_real, B_feats_fake, feat_criterion )

    gen_loss_A_total = (gen_loss_B*0.1 + fm_loss_B*0.9) * (1.-rate) +
recon_loss_A * rate
    gen_loss_B_total = (gen_loss_A*0.1 + fm_loss_A*0.9) * (1.-rate) +
recon_loss_B * rate
```

最后，我们对两个域 A 和 B 的损失进行加和，计算出 DiscoGAN 模型的总损失：

```
    if args.model_arch == 'discogan':
        gen_loss = gen_loss_A_total + gen_loss_B_total
        dis_loss = dis_loss_A + dis_loss_B
```

4.3 利用 PyTorch 从边框生成手提包

在本例中，我们将从 Berkley 的 pix2pix 数据集中利用手提包相关的边框图像生成逼真的手提包。请确保在进行下面的步骤前你的机器上已经安装了 PyTorch（http://pytorch.org/）和 OpenCV（http://docs.opencv.org/trunk/d7/d9f/

tutorial_linux_install.html)。

1. 首先克隆下面的代码仓库并进入 DiscoGAN 目录：

```
git clone https://github.com/SKTBrain/DiscoGAN.git
cd DiscoGAN
```

2. 接下来利用下面的命令下载 edges2handbags 数据集：

```
python ./datasets/download.py edges2handbags
```

3. 利用下载的数据集进行边框和手提包两个域的图像转换（见图 4-7）：

```
python ./discogan/image_translation.py --task_
name='edges2handbags'
```

```
ubuntu@ip-172-31-6-47:~/software/kuntalg/DiscoGAN$ python ./discogan/image_translation.py --task_name='edges2handbags'
/usr/local/lib/python2.7/dist-packages/torch/nn/functional.py:767: UserWarning: Using a target size (torch.Size([64, 1])) that is different
to the input size (torch.Size([64, 1, 1, 1])) is deprecated. Please ensure they have the same size.
  "Please ensure they have the same size.".format(target.size(), input.size()))
--------------------
GEN Loss: [ 0.63040555] [ 0.6074481]
Feature Matching Loss: [ 0.16809754] [ 0.2558834]
RECON Loss: [ 0.22831446] [ 0.169515]
DIS Loss: [ 0.69806194] [ 0.74758661]
Epoch #0| 1%|##                                                                                    |ETA:  0:18:14
```

图4-7：启动图像转换训练

4. 现在，每经过 1000 次迭代会在 results 目录下保存生成的图像，以之前每一步转换的任务名进行命名（见图 4-8）：

```
ubuntu@ip-172-31-6-47:~/software/kuntalg/DiscoGAN/results$ ls
edges2handbags  facescrub
```

图4-8：results目录示例

下面是一些从 *A* 到 *B* 的跨域生成图像输出样本（见图 4-9）。

图4-9：左侧为*A->B->A*（边框->手提包->边框）跨域转换生成的图像，
右侧为*B->A->B*（手提包->边框->手提包）跨域转换生成的图像

4.4 利用 PyTorch 进行性别转换

在本例中，我们会利用 `facescrub` 数据集中的名人图像来进行演员性别的相互转换。和前面相同，请确保在进行下面的步骤前你的机器上已经安装了 PyTorch 和 OpenCV。

1. 克隆下面的代码仓库并进入 `DiscoGAN` 目录（如果你在前面执行过这一步骤，则可以跳过）：

```
git clone https://github.com/SKTBrain/DiscoGAN.git
cd DiscoGAN
```

2. 通过下面的命令下载 `facescrub` 数据集：

```
python ./datasets/download.py facescrub
```

3. 利用下载的数据进行男性、女性之间的图像转换（见图 4-10）：

```
python ./discogan/image_translation.py --task_name= facescrub
```

```
ubuntu@ip-172-31-6-47:~/software/kuntalg/DiscoGAN$ python ./discogan/image_translation.py --task_name='
facescrub'
/usr/local/lib/python2.7/dist-packages/torch/nn/functional.py:767: UserWarning: Using a target size (to
rch.Size([64, 1])) that is different to the input size (torch.Size([64, 1, 1, 1])) is deprecated. Pleas
e ensure they have the same size.
  "Please ensure they have the same size.".format(target.size(), input.size()))
----------------------
GEN Loss: [ 0.74386597] [ 0.65673745]
Feature Matching Loss: [ 0.22089967] [ 0.2461448]
RECON Loss: [ 0.0708608] [ 0.0696818]
DIS Loss: [ 0.70826983] [ 0.70809972]
/usr/local/lib/python2.7/dist-packages/torch/nn/functional.py:767: UserWarning: Using a target size (to
rch.Size([63, 1])) that is different to the input size (torch.Size([63, 1, 1, 1])) is deprecated. Pleas
e ensure they have the same size.
  "Please ensure they have the same size.".format(target.size(), input.size()))
/usr/local/lib/python2.7/dist-packages/torch/nn/functional.py:767: UserWarning: Using a target size (to
rch.Size([62, 1])) that is different to the input size (torch.Size([62, 1, 1, 1])) is deprecated. Pleas
e ensure they have the same size.
  "Please ensure they have the same size.".format(target.size(), input.size()))
Epoch #0|  3%|##                                                          |ETA:  0:09:10
```

图4-10：启动图像转换训练

4. 现在每 1000 次迭代（依据 `image_save_interval` 不同而不同）后的生成图像会根据任务名（`facescrub`）和阶段间隔保存到 `results` 目录下（见图 4-11）：

```
[ubuntu@ip-172-31-6-47:~/software/kuntalg/DiscoGAN/results/facescrub/discogan$ ls
 0  1  2
```

图4-11：`results`目录示例

下面是一些从域 *A*（男性）到域 *B*（女性）跨域生成的图像样本输出（见图 4-12）。

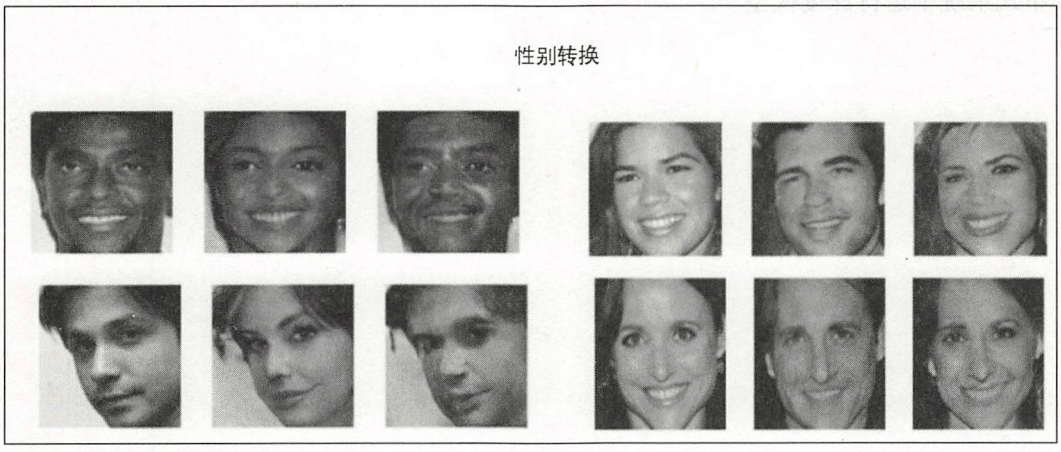


图4-12：左侧为*A*->*B*->*A*（男性->女性->男性）跨域转换生成的图像，
右侧为*B*->*A*->*B*（女性->男性->女性）跨域转换生成的图像

4.5 DiscoGAN 和 CycleGAN 的对比

本章提到的 DiscoGAN 和在第 2 章中提到的 CycleGAN 相比，提出了一种全新的不依赖域之间的关联信息，就可以完成寻找到域 A 到域 B 的映射关系并进行图像转换的方法。

从架构的角度来看，两个模型都包含了域之间相互映射的两个 GAN，并且都组合了传统的 GAN 损失和重构或者循环一致两种损失作为最终的损失。

两个模型之间并没有十分明显的不同，只是 DiscoGAN 利用两个重构损失来衡量经过 $X->Y->X$ 转换之后原始图像的质量，而 CycleGAN 使用一个拥有两个转换器 F 和 G（F 将域 A 的图像转换到域 B，G 做相反的转换）的循环一致损失来确保两个平衡点（$F(G(\boldsymbol{b})) = \boldsymbol{b}$ 和 $G(F(\boldsymbol{a})) = \boldsymbol{a}$，其中 $\boldsymbol{a}$ 和 $\boldsymbol{b}$ 分别为域 X 和域 Y 内的图像）是稳定的。

4.6 总结

到目前为止，你已经学会了组合多个 GAN 模型利用 StackGAN 和 DiscoGAN 来解决现实世界中复杂问题（从文本制作图像以及探索跨域关系）的方法。在第 5 章中我们会学习利用预训练模型和特征转换，针对小数据集进行深度学习的重要方法，以及如何在大规模分布式系统中运行深度模型。

5 利用多种生成模型生成图像

深度学习在更大的数据和更深的模型下能发挥更大的力量。这就导致了我们可能需要花数周的时间来训练上百万个参数。在现实生活中的场景下，我们可能并没有足够的数据、足够的硬件资源去训练更大规模的网络来达到预期的准确性。有没有办法能让我们无须每次从零开始来重新训练整个模型呢？

在本章中，我们首先通过一个真实数据集中的例子（MNIST，对车、猫、狗和花进行分类）介绍在如今深度学习应用里广泛使用且威力强大的**迁移学习（Transfer Learning）**。同时我们还会介绍利用 Apache Spark 和 BigDL 来在大规模分布式系统中构建深度学习网络。之后我们会结合迁移学习和 GAN 来生成高清晰度的逼真人脸图像。最后我们将了解如何通过 GAN 来生成具有幻觉感的艺术图像。

本章将会包含以下内容：

- 什么是迁移学习，它的优点及应用。
- 在 Keras 上利用预先训练的 VGG 模型对车、狗和花进行分类。
- 利用 Apache Spark 作为深度学习流水线在大规模分布式系统上部署和训练深度网络。
- 在 BigDL 上利用特征提取和微调来识别手写字母。
- 利用预训练的模型和 SRGAN 生成高清晰度图像。
- 利用 DeepDream 和 VAE 生成具有幻觉感的艺术图像。

从零开始构建深度学习网络需要大量的资源和时间。因此，我们通常并不想从零开始构建深度网络来解决手头的问题。我们通常会重用一个在类似问题上已经存在的模型并进行修改而不是重新制作“轮子”。

如果你想制造自动驾驶汽车，既可以花费数年时间来从零构建一个合适的图像识别算法，也可以利用 Google 从 ImageNet 大量图像中已经训练出来的 inception 模型。一个预训练的模型有可能并不满足你的精度需求，但却可以节省大量重新制作“轮子”的精力。通过一些微调和技巧，模型的准确度可以获得大幅的提升。

5.1 迁移学习介绍

预训练的模型通常情况下并不是为某个独特的数据集而设计的，但是其对于启动处理手头类似的任务十分有帮助。

举例来说，InceptionV3 作为一个十分流行的模型可以对 1000 个类别的图像进行分类，但对于我们的问题来说只想将狗识别出来。在深度学习领域将一个已有的训练模型用于处理手头上另一个类似任务的方法被称为迁移学习。

这也是最近几年迁移学习在深度学习从业者中使用得越来越广泛，并且也成为很多现实问题的解决方案的原因。它是一种将知识（特征）在不同域之间转换的技术。

5.1.1 迁移学习的目的

假设我们已经训练好了一个可以区分新鲜杧果和腐烂杧果的模型。在训练网络的过程中我们已经使用了数千张腐烂和新鲜的杧果图像，并花费数小时进行训练来学习这些水果是否腐烂、是否有液体流出、是否产生难闻的气味这些知识。现在我们可以把这个已经训练好的网络用于新的场景。例如，通过从杧果图像中学习到的关于腐烂的知识来区分腐烂和新鲜的苹果。

迁移学习的通常方法是先训练一个基本网络，然后将网络的前 n 层复制到目标网络的前 n 层。目标网络的其他层被随机地初始化，并向着我们目标场景的方向进行训练。

迁移学习主要应用于深度学习的下面几个场景。

- **小数据集：**如果你的数据集很小，那么从零开始构建深度网络通常无法奏效。迁移学习提供了将预训练的模型在新类型数据集上进行应用的方法。具体来说，一个在百万张图像数据集 ImageNet 上预先训练好的模型，在一个新的小数据集上（CIFAR-10 的部分数据）训练的效果会优于从同样数据集上从零训练的深度模型。
- **少量的资源：**深度学习的处理过程（例如卷积）需要消耗大量的时间和资源。深度学习的训练过程十分适合用高端 GPU 的机器来处理。然而利用预训练模型，由于

大部分修改只在最后一层进行分类或者回归的权重更新，因此你可以轻松地在一台没有 GPU 的笔记本电脑中花费不到一分钟的时间处理全量的训练集（比如 50 000 张图像）。

5.1.2 多种利用预训练模型的方法

接下来我们会讨论多种使用预训练模型的方法。

- **使用预处理架构：**除了利用训练模型的权重外，我们还可以只利用架构来在新的数据上进行训练。
- **特征提取：**一个预训练模型可以被用来进行特征提取。我们只需要移除预训练网络的输出层，并将剩余层冻结，作为新数据集固定的提取特征。
- **网络部分冻结：**除了仅仅替换最终层并从之前所有层提取特征外，我们还可以通过部分层来训练模型；也就是保持初始几层的权重不变，重新训练其他剩余的层。我们可以通过一个或多个超参数来控制冻结哪些层（见图 5-1）。

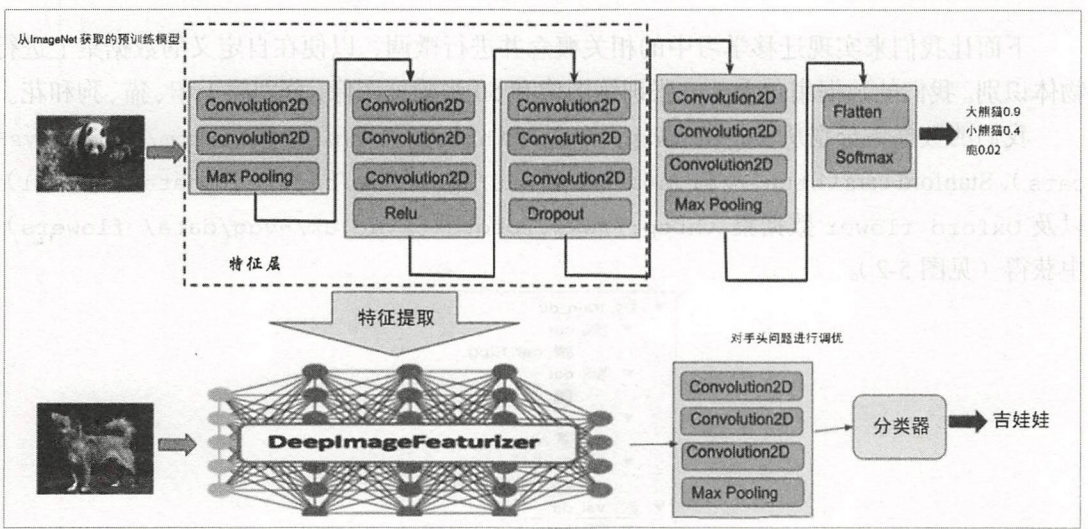


图5-1：利用预训练模型进行迁移学习

你需要根据数据集的大小和数据集的相似度来调整迁移学习的过程。表 5-1 中是一些场景。

表 5-1　不同场景下的训练方案

	高数据相似度	低数据相似度
小数据集	在小数据集但高相似度的场景下，我们仅需要修改预训练模型的输出层，并将其作为特征提取器	如果数据集小且相似度低，我们可以冻结预训练网络的前 k 层，训练剩下的 $n-k$ 层。这样我们可以定制初始化层，小数据集也可以通过冻结初始 k 层获得补偿
大数据集	在这个场景中我们可以复用预训练模型的架构和初始权重	尽管我们有大量数据，但是数据和预训练模型的数据集差异十分大，因此在这种场景中迁移学习的效果不会太好，最好还是从零开始训练深度网络

在图像识别的场景中，迁移学习利用预训练模型的卷积层来提取新输入图像的特征，这就意味着只有小部分的原始模型需要重新训练，网络的其他部分保持不变。通过这种方法，我们节省了大量的资源和时间，原始图像只需要经过冻结网络部分一次即可。

5.1.3　利用 Keras 对车、猫、狗和花进行分类

下面让我们来实现迁移学习中的相关概念并进行微调，以便在自定义的数据集上进行物体识别。我们的数据集包含 150 张训练图像和 50 张验证图像，分别属于车、猫、狗和花。

我们的数据集分别从 *Kaggle Dogs vs. Cats*（`https://www.kaggle.com/c/dogs-vs-cats`）、Stanford cars（`http://ai.stanford.edu/~jkrause/cars/car_dataset.html`）以及 `Oxford flower` 数据集（`http://www.robots.ox.ac.uk/~vgg/data/ flowers`）中获得（见图 5-2）。

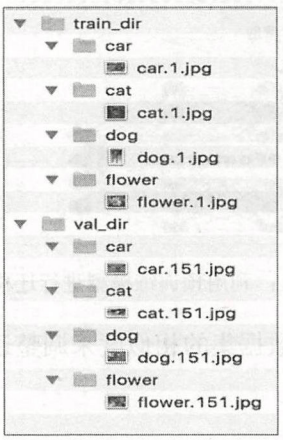


图5-2：车、猫、狗和花的数据集结构

我们利用 preprocessing 函数对图像进行一些预处理，接下来应用 rotation、shift、shear、zoom 和 flip 参数对数据进行放大和转换：

```
train_datagen = ImageDataGenerator(
    preprocessing_function=preprocess_input,
    rotation_range=30,
    width_shift_range=0.2,
    height_shift_range=0.2,
    shear_range=0.2,
    zoom_range=0.2,
    horizontal_flip=True
)

test_datagen = ImageDataGenerator(
    preprocessing_function=preprocess_input,
    rotation_range=30,
    width_shift_range=0.2,
    height_shift_range=0.2,
    shear_range=0.2,
    zoom_range=0.2,
    horizontal_flip=True
)

train_generator = train_datagen.flow_from_directory(
  args.train_dir,
  target_size=(IM_WIDTH, IM_HEIGHT),
  batch_size=batch_size,
)

validation_generator = test_datagen.flow_from_directory(
  args.val_dir,
  target_size=(IM_WIDTH, IM_HEIGHT),
  batch_size=batch_size,
)
```

接下来我们需要从 keras.applications 模块载入 InceptionV3 模型。我们利用 include_top=False 参数来保留最后几个全连接层的权重：

```
base_model = InceptionV3(weights='imagenet', include_top=False)
```

通过加入一个 1024 大小的全连接层来初始化最后一层，在输出层利用 softmax 函数将结果固定在[0,1]之间：

```
def addNewLastLayer(base_model, nb_classes):
  x = base_model.output
  x = GlobalAveragePooling2D()(x)
  x = Dense(FC_SIZE, activation='relu')(x)
  predictions = Dense(nb_classes, activation='softmax')(x)
  model = Model(input=base_model.input, output=predictions)
  return model
```

一旦最后一层固定（迁移学习），我们就可以继续训练更多的层（微调）。

利用下面的工具方法来冻结所有层并编译模型：

```
def setupTransferLearn(model, base_model):
  for layer in base_model.layers:
    layer.trainable = False
  model.compile(optimizer='rmsprop',
                loss='categorical_crossentropy',
                metrics=['accuracy'])
```

下面是另一个工具方法，用于冻结 InceptionV3 中最上面的两个 inception 处理层，然后重新训练其他层：

```
def setupFineTune(model):
   for layer in model.layers[:NB_IV3_LAYERS_TO_FREEZE]:
      layer.trainable = False
   for layer in model.layers[NB_IV3_LAYERS_TO_FREEZE:]:
      layer.trainable = True
   model.compile(optimizer=SGD(lr=0.0001, momentum=0.9),
loss='categorical_crossentropy')
```

下面可以利用 fit_generator 方法进行训练，并保存最后的模型：

```
history = model.fit_generator(
  train_generator,
  samples_per_epoch=nb_train_samples,
  nb_epoch=nb_epoch,
```

```
    validation_data=validation_generator,
    nb_val_samples=nb_val_samples,
    class_weight='auto')
model.save(args.output_model_file)
```

运行下面的命令进行训练和微调（见图 5-3）：

```
python training-fine-tune.py --train_dir <path to training images> --val_dir
<path to validation images>
```

```
ubuntu@ip-172-31-1-246:~/software/dataset/cat-dog-flower-car$ python training-fine-tune.py --train_dir /home/ubuntu/software/dataset/cat-
dog-car-flower/train_dir --val_dir /home/ubuntu/software/dataset/cat-dog-car-flower/val_dir

Using TensorFlow backend.
Found 600 images belonging to 4 classes.
Found 200 images belonging to 4 classes.
Downloading data from https://github.com/fchollet/deep-learning-models/releases/download/v0.2/inception_v3_weights_tf_dim_ordering_tf_ker
```

图5-3：启动模型训练

即使我们的数据集很小，通过利用预训练模型和迁移学习，依然在验证集上达到了 98.5% 的准确率（见图 5-4）。

```
Epoch 1/3
128/600 [=====>........................] - ETA: 15s - loss: 0.0424 - acc: 0.98442017-08-13 07:06:31.582692: I tensorflow/core/common_runt
ime/gpu/pool_allocator.cc:247] PoolAllocator: After 11762 get requests, put_count=11717 evicted_count=1000 eviction_rate=0.0853461 and un
satisfied allocation rate=0.0973474
2017-08-13 07:06:31.582730: I tensorflow/core/common_runtime/gpu/pool_allocator.cc:259] Raising pool_size_limit_ from 100 to 110
600/600 [==============================] - 15s - loss: 0.0522 - acc: 0.9833 - val_loss: 0.0708 - val_acc: 0.9800
Epoch 2/3
600/600 [==============================] - 13s - loss: 0.0424 - acc: 0.9883 - val_loss: 0.0657 - val_acc: 0.9850
Epoch 3/3
600/600 [==============================] - 13s - loss: 0.0240 - acc: 0.9933 - val_loss: 0.1065 - val_acc: 0.9850
```

图5-4：训练结果示例

现在我们可以利用保存的模型对本地文件系统或者 URL 里的图像进行图像预测了（见图 5-5）：

```
python predict.py --image_url https://goo.gl/DCbuq8 --model
inceptionv3-ft.model
```

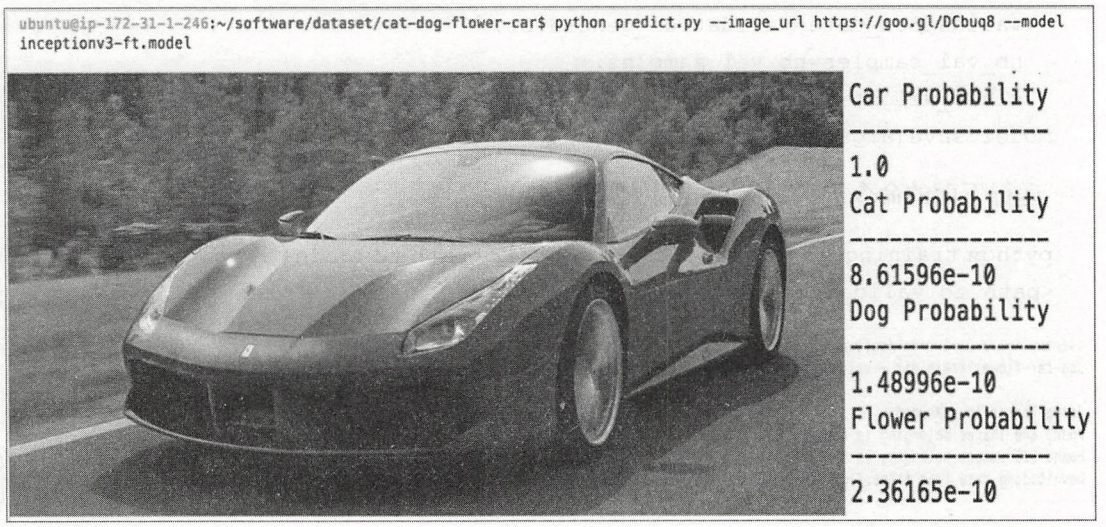


图5-5：对车进行分类预测的结果

5.2 利用 Apache Spark 进行大规模深度学习

深度学习是一个极度消耗资源并且计算密集型的任务，你可以通过更多的数据和更大的网络来获得更好的结果，但是它的训练速度也受数据集大小的影响。在现实世界中，深度学习需要尝试调整各种参数（也被称为超参数）来进行实验，因此需要在大数据集上迭代运行多次深度网络，速度是一个很重要的因素。一些常用的办法包括更快的硬件（GPU）、优化代码（利用合适的生产环境框架），以及在分布式集群上进行扩展来获得并行加速的能力。

数据并行的核心思想是将大数据集切分成多个数据块，将这些数据块在分布式集群的节点上分别进行神经网络的处理。

Apache Spark 是一个高速、通用、容错的处理交互式或迭代式任务的大规模分布式内存计算框架，它主要处理内存中的 RDD 或 DataFrame 而不是将数据保存在磁盘上。它支持包括存储层在内的多个数据源，并屏蔽了不同的数据格式和数据流而提供统一的数据访问方式，在之上定义了更高层次的复杂和可组合操作。

现如今 Spark 已经成为大数据处理和大数据访问的“超级武器”，但是仅仅依赖 Spark 的核心模块并不能在集群上训练深度网络。在下面几节中，我们会利用优化过的库在 Apache Spark 上开发深度学习应用。

出于编码的目的，我们只运行单机模式的 Apache Spark，并不会介绍如何搭建分布式 Spark 集群。关于 Spark 集群模式的信息请参考：https://spark.apache.org/docs/latest/cluster-overview.html 。

5.2.1 利用 Spark 深度学习模块运行预训练模型

深度学习流水线是一个可以帮助开发人员以简单的方式来释放 Apache Spark 集群运行大规模深度学习潜力的开源库。它利用 Apache Spark ML Pipeline 进行模型训练，利用 Spark DataFrame 和 SQL 来进行模型部署。它提供了一组更高层级的 API 来使用 Spark ML Pipeline 中预训练的模型在分布式系统中运行迁移学习。

深度学习流水线利用 featurizer 的概念使得迁移学习更简单。featurizer（或者针对图像操作的 `DeepImageFeaturizer`）自动地移除预训练神经网络模型的最后一层，并利用之前所有层的输出作为针对新领域问题分类算法（例如逻辑回归）的特征。

下面让我们实现深度学习流水线，以在 Spark 集群上利用预训练的模型来从花类的数据集（`http://download.tensorflow.org/example_images/flower_photos.tgz`）进行预测。

1. 启动 PySpark 的深度学习流水线：

```
pyspark --master local[*] --packages
databricks:spark-deep-learning:0.1.0-spark2.1-s_2.11
```

注意：如果你在启动 PySpark 深度学习流水时发现 **No module name sparkdl**，请参考下面的 GitHub 页面来解决：https://github.com/databricks/spark-deep-learning/issues/18

2. 读取图像，然后随机地将图像分为训练（`train`）集和测试（`test`）集：

```
img_dir= "path to base image directory"
roses_df = readImages(img_dir + "/roses").withColumn("label", lit(1))
daisy_df = readImages(img_dir + "/daisy").withColumn("label", lit(0))
roses_train, roses_test = roses_df.randomSplit([0.6, 0.4])
daisy_train, daisy_test = daisy_df.randomSplit([0.6, 0.4])
```


```
train_df = roses_train.unionAll(daisy_train)
```

3. 利用 InceptionV3 模型和 DeepImageFeaturizer 创建流水线：

```
featurizer = DeepImageFeaturizer(inputCol="image", outputCol="features",
modelName="InceptionV3")
lr = LogisticRegression(maxIter=20, regParam=0.05, elasticNetParam=0.3,
labelCol="label")
p = Pipeline(stages=[featurizer, lr])
```

4. 利用已存在的预训练模型来对图像进行适配，train_images_df 是图像和标签的数据集：

```
p_model = p.fit(train_df)
```

5. 最终我们计算该模型的准确性（见图 5-6）：

```
tested_df = p_model.transform(test_df)
evaluator = MulticlassClassificationEvaluator(metricName="accuracy")
print("Test set accuracy = " +
str(evaluator.evaluate(tested_df.select("prediction", "label"))))
```

```
>>> evaluator = MulticlassClassificationEvaluator(metricName="accuracy")
>>> print("Test set accuracy = " + str(evaluator.evaluate(tested_df.select("prediction", "label"))))
Test set accuracy = 0.962264150943
```

图5-6：计算模型的准确性

除了使用 DeepImageFeaturizer 之外，我们也可以直接用预训练模型进行预测，并不需要任何的重新训练和微调：

```
sample_img_dir=<path to your image>

image_df = readImages(sample_img_dir)

predictor = DeepImagePredictor(inputCol="image",
outputCol="predicted_labels", modelName="InceptionV3", decodePredictions=True,
topK=10)
predictions_df = predictor.transform(image_df)

predictions_df.select("filePath", "predicted_labels").show(10,False)
```

输入图像和预测的前 5 名如图 5-7 所示。

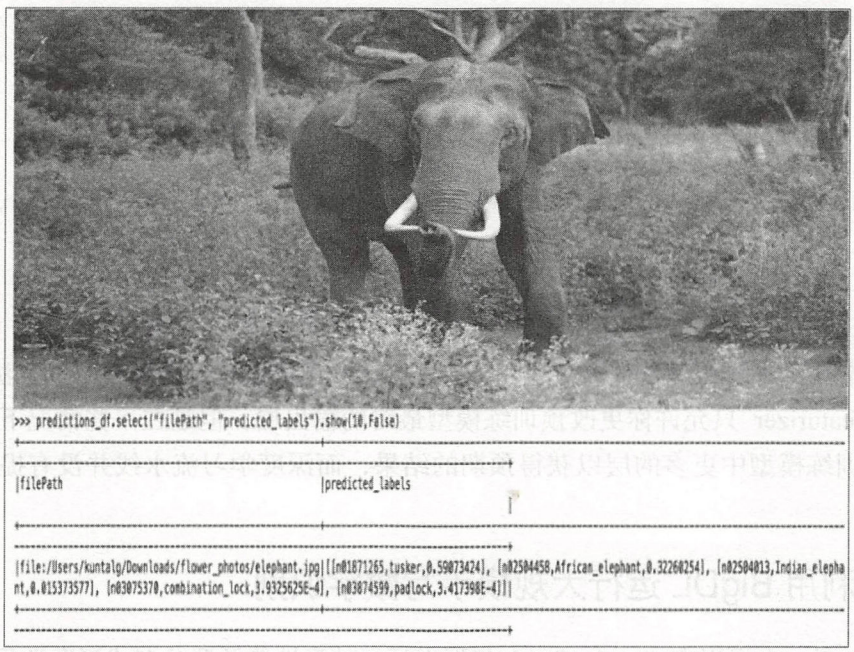


图5-7：对大象类别进行预测的结果

除了使用内置的预训练模型外，深度学习流水线还允许用户使用从 Keras 或者 TensorFlow 训练的模型。这就使得一个单机的深度模型可以部署到大规模分布式系统中进行大数据量的训练。

首先我们载入 Keras 内置的 InceptionV3 模型，并保存成文件：

```
model = InceptionV3(weights="imagenet")
model.save('model-full.h5')
```

随后，在预测阶段我们载入模型，并将图像传入模型来得到所期望的预测结果：

```
def loadAndPreprocessKerasInceptionV3(uri):
    # this is a typical way to load and prep images in keras
    image = img_to_array(load_img(uri, target_size=(299, 299)))
    image = np.expand_dims(image, axis=0)
    return preprocess_input(image)
```

```
    transformer = KerasImageFileTransformer(inputCol="uri",
      outputCol="predictions", modelFile="model-full.h5", imageLoader=
      loadAndPreprocessKerasInceptionV3, outputMode="vector")
    dirpath=<path to mix-img>

    files = [os.path.abspath(os.path.join(dirpath, f)) for f in
os.listdir(dirpath) if f.endswith('.jpg')]
    uri_df = sqlContext.createDataFrame(files, StringType()).toDF("uri")

    final_df = transformer.transform(uri_df)
    final_df.select("uri", "predictions").show()
```

深度学习流水线的确是一个快速在分布式 Spark 集群上运行迁移学习的方法。需要注意的是，featurizer 只允许你更改预训练模型最终的输出层，但是在一些情况下你可能希望修改预训练模型中更多的层以获得预期的结果，而深度学习流水线并没有提供这方面的能力。

5.2.2 利用 BigDL 运行大规模手写数字识别

BigDL 是一个可以在 Apache Spark 集群上运行的高性能开源分布式深度学习库。其通过利用 Intel Math Kernel Library（MKL）以及在每个 Spark 任务内使用多线程来达到高性能。BigDL 提供了 Keras 风格（串行式以及函数式）的高层次 API 来构建深度学习应用，并且可以很好地扩展到更大规模的分析处理中。使用 BigDL 的主要动机如下：

- 在大规模分布式系统中运行深度学习模型以及分析已经存在于 Spark 或者 Hadoop（Hive、HDFS 或者 HBase）中的数据。
- 给现有的大数据工作流增加深度学习相关功能（训练和预测）。

从图 5-8 中可以看到，BigDL 驱动程序首先在集群中的主节点启动。接下来通过**集群管理器**和驱动程序的帮助，Spark 任务被分散到集群的每个工作节点。BigDL 通过和 Intel MKL 交互使得这些任务可以更快速地执行。

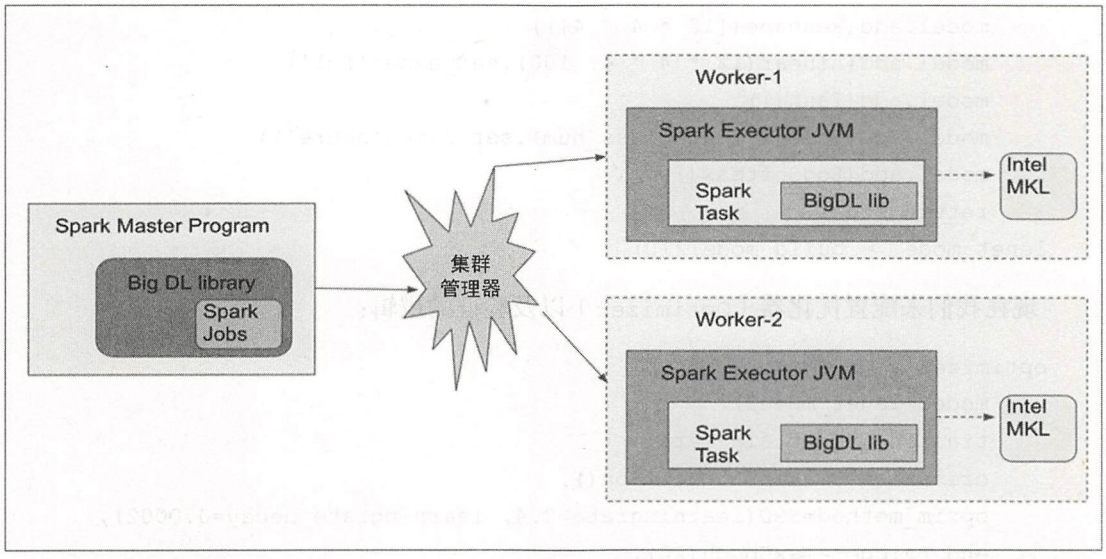


图5-8：在Spark集群上执行BigDL

接下来让我们在大规模数据集上实现一个深度神经网络，以在 mnist 数据集上识别手写的数字。首先我们准备训练和验证样本：

```
mnist_path = "datasets/mnist"
(train_data, test_data) = get_mnist(sc, mnist_path)
print train_data.count()
print test_data.count()
```

之后我们完成一个 LeNet 架构，包括两组卷积层、一个激发层和一个池化层，之后是一个全连接层、一个激发层以及另一个全连接层，最后是一个 SoftMax 分类器。LeNet 网络很小，但是足够给我们提供有趣的结果：

```
def build_model(class_num):
    model = Sequential()
    model.add(Reshape([1, 28, 28]))
    model.add(SpatialConvolution(1, 6, 5, 5).set_name('conv1'))
    model.add(Tanh())
    model.add(SpatialMaxPooling(2, 2, 2, 2).set_name('pool1'))
    model.add(Tanh())
    model.add(SpatialConvolution(6, 12, 5, 5).set_name('conv2'))
    model.add(SpatialMaxPooling(2, 2, 2, 2).set_name('pool2'))
```

```
    model.add(Reshape([12 * 4 * 4]))
    model.add(Linear(12 * 4 * 4, 100).set_name('fc1'))
    model.add(Tanh())
    model.add(Linear(100, class_num).set_name('score'))
    model.add(LogSoftMax())
    return model
lenet_model = build_model(10)
```

现在我们来配置优化器（Optimizer）以及验证的逻辑：

```
optimizer = Optimizer(
    model=lenet_model,
    training_rdd=train_data,
    criterion=ClassNLLCriterion(),
    optim_method=SGD(learningrate=0.4, learningrate_decay=0.0002),
    end_trigger=MaxEpoch(20),
    batch_size=2048)

optimizer.set_validation(
    batch_size=2048,
    val_rdd=test_data,
    trigger=EveryEpoch(),
    val_method=[Top1Accuracy()]
)

trained_model = optimizer.optimize()
```

接下来我们从中选取几个测试样本，检测预测的标签和真实情况来进行预测。

```
predictions = trained_model.predict(test_data)
```

最后我们利用 spark-submit 命令来在 Spark 集群上训练 LeNet 模型。下载 BigDL 在 Apache Spark 上的发行版（https://bigdl-project.github.io/master/#release-download/），接下来执行代码中的 run.sh 文件将任务提交给 Spark 集群：

```
SPARK_HOME= <path to Spark>
BigDL_HOME= <path to BigDL>
PYTHON_API_ZIP_PATH=${BigDL_HOME}/bigdl-python-<version>.zip
BigDL_JAR_PATH=${BigDL_HOME}/bigdl-SPARK-<version>.jar
```

```
export PYTHONPATH=${PYTHON_API_ZIP_PATH}:${BigDL_HOME}/conf/spark-
bigdl.conf:$PYTHONPATH

${SPARK_HOME}/bin/spark-submit \
    --master <local or spark master url>\
    --driver-cores 5  \
    --driver-memory 5g  \
    --total-executor-cores 16  \
    --executor-cores 8  \
    --executor-memory 10g \
    --py-files ${PYTHON_API_ZIP_PATH},${BigDL_HOME}/BigDL-MNIST.py\
    --properties-file ${BigDL_HOME}/conf/spark-bigdl.conf \
    --jars ${BigDL_JAR_PATH} \
    --conf spark.driver.extraClassPath=${BigDL_JAR_PATH} \
    --conf spark.executor.extraClassPath=bigdl-SPARK<version>.jar \
    ${BigDL_HOME}/BigDL-MNIST.py
```

更多关于 spark-submit 的信息可以通过下面的链接来获得：https://spark.apache.org/docs/latest/submitting-applications.html。

当你提交了任务之后，可以通过 Spark Master 节点的页面来了解执行情况（见图 5-9）。

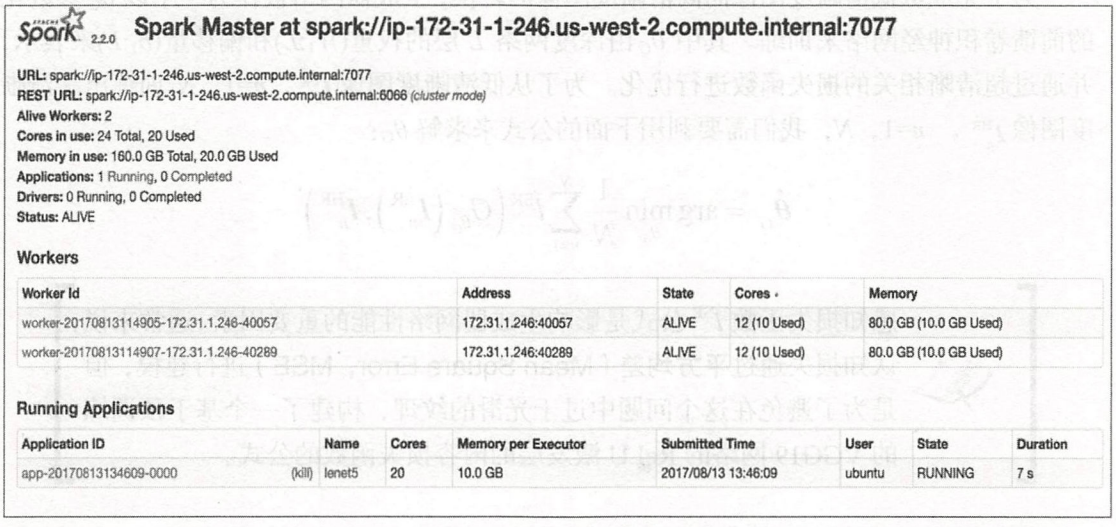


图5-9：BigDL LeNet5模型任务运行在Apache Spark集群上

当任务成功结束后，你可以搜索 Spark 工作节点上的日志来验证模型的准确度，结果应该和下面相近：

```
INFO  DistriOptimizer$:536 - Top1Accuracy is Accuracy(correct: 9568,
count: 10000, accuracy: 0.9568)
```

5.2.3 利用 SRGAN 生成高清晰度图像

超清晰生成网络（Super Resolution Generative Network，SRGAN）在从低清晰度图像生成高清晰度图像方面十分优秀。在训练的过程中，一张高清晰度图像经过高斯过滤转换成一张低清晰度图像，接下来通过向下采样变成一张高清晰度图像。

在我们进入网络架构细节前先介绍一些定义和术语。

- I^{LR}：宽（W）×高（H）×颜色通道（C）大小的低清晰度图像。
- I^{HR}：$W \times H \times C$ 大小的高清晰度图像。
- I^{SR}：$W \times H \times C$ 大小的超清晰度图像。
- r：向下采样因子。
- G_{θ_G}：生成器网络。
- D_{θ_D}：判别器网络。

为了完成从低清晰度图像向高清晰度图像的转换，生成器网络被作为一个以 θ_G 参数化的前馈卷积神经网络来训练。其中 θ_G 由深度网络 L 层的权重(W_1:L)和偏移量(b_1:L)来表示，并通过超清晰相关的损失函数进行优化。为了从低清晰度图像 I_n^{LR}，n=1，N 训练出高清晰度图像 I_n^{HR}，n=1，N，我们需要利用下面的公式来求解 θ_G：

$$\hat{\theta}_G = \arg\min_{\theta_G} \frac{1}{N} \sum_{n=1}^{N} l^{SR}\left(G_{\theta_G}\left(I_n^{LR}\right), I_n^{HR}\right)$$

感知损失函数 l^{SR} 公式是影响生成器网络性能的重要因素。通常来说，认知损失通过**平方均差（Mean Square Error，MSE）**进行建模，但是为了避免在这个问题中过于光滑的纹理，构建了一个基于预训练的 VGG19 网络的 ReLU 激发层的内容损失函数的公式。

感知损失由多个对超清晰图像生成相关的损失函数组合而成：

$$l^{SR} = \underbrace{\underbrace{l^{SR}_{VGG/i.j}}_{\text{内容损失}} + \underbrace{10^{-3}l^{SR}_{Gen}}_{\text{对抗损失}}}_{\text{感知损失（基于VGG的内容损失）}}$$

- **内容损失**：基于 VGG 的内容损失通过重构图像 G_{θ_G} 的特征表示和对应的高清晰度图像 I^{HR} 之间的欧氏距离表示。在这里 $\Phi_{i,j}$ 表示在 VGG19 网络中第 i 个最大值池化层前第 j 个卷积层的特征图。$W_{i,j}$、$H_{i,j}$ 表示 VGG 网络中对应特征图的维度：

$$l^{SR}_{VGG/i.j} = \frac{1}{W_{i,j}H_{i,j}}\sum_{x=1}^{W_{i,j}}\sum_{y=1}^{H_{i,j}}\left(\varnothing_{i,j}\left(I^{HR}\right)x,y - \varnothing_{i,j}\left(G_{\theta_G}\left(I^{LR}\right)\right)x,y\right)^2$$

- **对抗损失**：生成损失基于在所有训练图像上生成器 $D_{\theta_D}(G_{\theta_G}(I^{LR}))$ 的概率，并且会奖赏那些判别器无法进行区分的方案：

$$l^{SR}_{Gen} = \sum_{n=1}^{N} -\log D_{\theta_D}\left(G_{\theta_G}\left(I^{LR}\right)\right)$$

和对抗网络的概念类似，SRGAN 的背后思想也是以欺骗判别器为目的进行训练，使得判别器无法区分高清晰度的生成图像和真实图像：

$$\min_{\theta_G}\max_{\theta_D}\mathbb{E}_{I^{HR}\sim p_{train}\left(I^{HR}\right)}\left[\log D_{\theta_D}\left(I^{HR}\right)\right] + \mathbb{E}_{I^{LR}\sim p_G\left(I^{LR}\right)}\left[\log\left(1-D_{\theta_D}\right)\left(G_{\theta_G}\left(I^{LR}\right)\right)\right]$$

通过这种方式生成器学会如何制造和真实图像高度相似，而判别器 D 无法进行区分的高清晰度图像。

5.2.4 SRGAN 的架构

如图 5-10 中的展示，生成器网络 G 包含 8 个有相同布局的残差块。每一块包含两个 3 × 3 核的卷积层、64 个特征图以及一个批规范化层，并使用 ParametricReLU 作为激发（activation）函数。输入图像的清晰度通过两个训练后的子像素卷积层进行提升。

判别器网络使用 Leaky ReLU（α=0.2）作为激发函数并包含 8 个过滤核不断增加的卷积层。核数以 2 的倍数从 64 核增长到 512 核。每次特征翻倍，就使用步长卷积来降低图像的清晰度。

最终的 512 特征图经过两个致密层以及最终的 sigmoid 激发层得到图像样本是真实图像的概率。

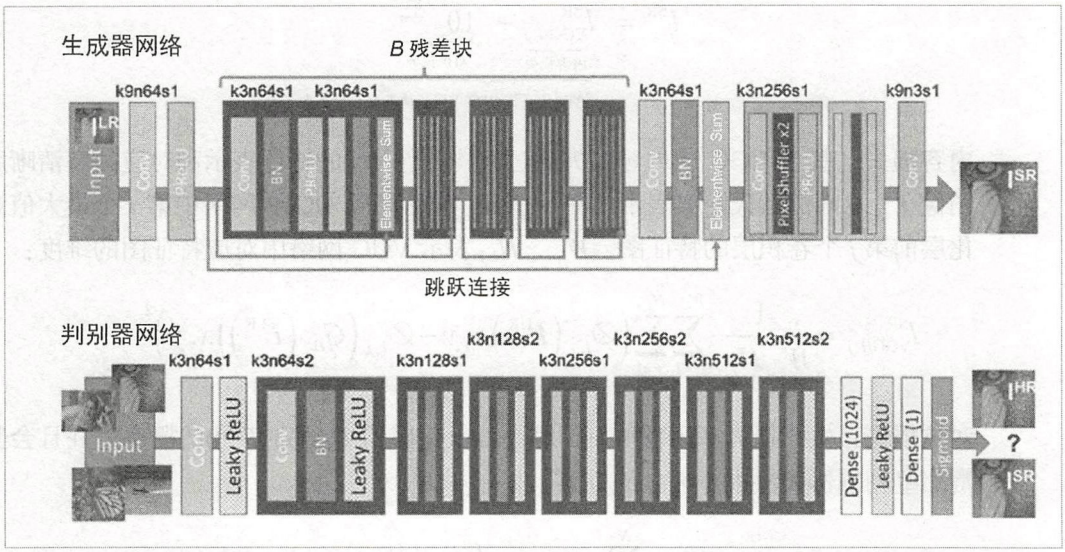


图5-10：生成器网络和判别器网络的架构，每一个卷积层的参数：核数（k）、特征图数（n）和步长（s）（来源：*arXiv: 1609.04802, 2017*）

现在是时候深入代码并在 TensorFlow 上利用 LFW 脸部数据集生成高清晰度图像了。

生成器网络首先是一个单独的 3 × 3 核的逆卷积层，接下来是 64 个特征图和 ReLU 为激发函数的激发层。再之后是 5 个拥有相同布局的残差块，每块拥有两个卷积层以及批规范化和 ReLU。最终输入图像的清晰度通过两个训练后的子像素层进行提升：

```
def generator(self, x, is_training, reuse):
    with tf.variable_scope('generator', reuse=reuse):
      with tf.variable_scope('deconv1'):
          x = deconv_layer(
              x, [3, 3, 64, 3], [self.batch_size, 24, 24, 64], 1)
          x = tf.nn.relu(x)
          shortcut = x
# 5个相同布局的残差块，拥有相同的逆卷积层和批规范化以及ReLU激发函数
    for i in range(5):
       mid = x
       with tf.variable_scope('block{}a'.format(i+1)):
         x = deconv_layer(x, [3, 3, 64, 64], [self.batch_size, 24,
24, 64], 1)
         x = batch_normalize(x, is_training)
```

```
        x = tf.nn.relu(x)

# 两个进行了pixel-suffle的逆卷积层，并用ReLU作为激发函数
    with tf.variable_scope('deconv3'):
        x = deconv_layer(x, [3, 3, 256, 64], [self.batch_size, 24,
24, 256], 1)
        x = pixel_shuffle_layer(x, 2, 64) # n_split = 256 / 2 ** 2
        x = tf.nn.relu(x)
    with tf.variable_scope('deconv4'):
        x = deconv_layer(x, [3, 3, 64, 64], [self.batch_size, 48,
48, 64], 1)
        x = pixel_shuffle_layer(x, 2, 16)
        x = tf.nn.relu(x)

    . . . . . .[code omitted for clarity]

return x
```

逆卷积函数通过 TensorFlow 中的 `conv2d_transpose` 方法和 Xavier 来初始化：

```
def deconv_layer(x, filter_shape, output_shape, stride, trainable=True):
    filter_ = tf.get_variable(
        name='weight',
        shape=filter_shape,
        dtype=tf.float32,
        initializer=tf.contrib.layers.xavier_initializer(),
        trainable=trainable)
    return tf.nn.conv2d_transpose(
        value=x,
        filter=filter_,
        output_shape=output_shape,
        strides=[1, stride, stride, 1])
```

判别器网络包括 8 个 3×3 过滤核不断增加的卷积层，核数以 2 的倍数从 64 到 512 增长。最终的 512 特征图扁平化后通过两个全连接层最终通过 softmax 层来获得图像是生成图像的概率：

```
def discriminator(self, x, is_training, reuse):
      with tf.variable_scope('discriminator', reuse=reuse):
```

```
            with tf.variable_scope('conv1'):
                x = conv_layer(x, [3, 3, 3, 64], 1)
                x = lrelu(x)
            with tf.variable_scope('conv2'):
                x = conv_layer(x, [3, 3, 64, 64], 2)
                x = lrelu(x)
                x = batch_normalize(x, is_training)

        . . . . . [code omitted for clarity]

            x = flatten_layer(x)
            with tf.variable_scope('fc'):
                x = full_connection_layer(x, 1024)
                x = lrelu(x)
            with tf.variable_scope('softmax'):
                x = full_connection_layer(x, 1)

return x
```

网络利用 Leaky ReLU（a=0.2）作为卷积层的激发函数：

```
def lrelu(x, trainbable=None):
    alpha = 0.2
    return tf.maximum(alpha * x, x)
```

卷积层函数通过 TensorFlow 的 conv2d 方法实现并用 Xavier 初始化：

```
def conv_layer(x, filter_shape, stride, trainable=True):
    filter_ = tf.get_variable(
        name='weight',
        shape=filter_shape,
        dtype=tf.float32,
        initializer=tf.contrib.layers.xavier_initializer(),
        trainable=trainable)
    return tf.nn.conv2d(
        input=x,
        filter=filter_,
        strides=[1, stride, stride, 1],
        padding='SAME')
```

请注意在代码的实现中判别器使用了最小二乘差损失函数来避免梯度消失的问题，而不是在原始 SRGAN 论文（*arXiv: 1609.04802, 2017*）中提到的 sigmoid 交叉熵损失函数：

```
def inference_adversarial_loss(real_output, fake_output):
alpha = 1e-5
g_loss = tf.reduce_mean(
tf.nn.l2_loss(fake_output - tf.ones_like(fake_output)))
d_loss_real = tf.reduce_mean(
tf.nn.l2_loss(real_output - tf.ones_like(true_output)))
d_loss_fake = tf.reduce_mean(
tf.nn.l2_loss(fake_output + tf.zeros_like(fake_output)))
d_loss = d_loss_real + d_loss_fake
return (g_loss * alpha, d_loss * alpha)

generator_loss, discriminator_loss = (
inference_adversarial_loss(true_output, fake_output))
```

更多最小二乘 GAN 的资料可以参考 https://arxiv.org/abs/1611.04076。

SRGAN 的代码结构如图 5-11 所示。

```
[ubuntu@ip-172-31-1-246:~/software/dataset/srgan/code$ ls -lth
total 52K
drwxrwxr-x 3 ubuntu ubuntu 4.0K Aug 13 14:10 backup
drwxr-xr-x 2 ubuntu ubuntu 4.0K Aug 13 14:08 result
drwxr-xr-x 5 ubuntu ubuntu 4.0K Aug 13 14:07 data
-rwxr-xr-x 1 ubuntu ubuntu 3.3K Aug 13 13:03 trainSrgan.py
-rwxr-xr-x 1 ubuntu ubuntu  150 Aug 13 12:36 loadLfw.py
-rwxr-xr-x 1 ubuntu ubuntu 4.2K Aug 13 12:14 vgg19.py
-rwxr-xr-x 1 ubuntu ubuntu 4.2K Aug 13 12:12 utilLayer.py
-rwxr-xr-x 1 ubuntu ubuntu 1.2K Aug 13 12:12 utilAugment.py
-rwxr-xr-x 1 ubuntu ubuntu 6.9K Aug 13 12:12 srgan.py
-rwxr-xr-x 1 ubuntu ubuntu 2.7K Aug 13 12:12 download-preprocess-lfw.py
```

图5-11：SRGAN代码结构目录

首先让我们下载 LFW 脸部数据集并做一些预处理（脸部检测以及切分训练集和测试集），记下来将数据保存在 data/目录下（见图 5-12）：

```
python download-preprocess-lfw.py
```

```
ubuntu@ip-172-31-1-246:~/software/dataset/srgan/code$ python download-preprocess-lfw.py
... loading data
100%|████████████████████████████████████████████████████████| 12577/12577 [11:45<00:00, 17.82it/s]
100%|████████████████████████████████████████████████████████| 662/662 [00:36<00:00, 17.92it/s]
```

图5-12：下载脸部数据集

接下来通过下面的链接下载 VGG19 预训练模型，解压缩并保存在 backup/目录下：

```
https://drive.google.com/open?id=0B-s6ok7B0V9vcXNfSzdjZ0lCc0k
```

下一步执行 trainSrgan.py 文件来利用 VGG19 模型启动 SRGAN 训练：

```
python trainSrgan.py
```

一旦训练开始，生成器网络就会在 result/目录下生成超清晰的图像，其中的一些样例图像如图 5-13 所示。

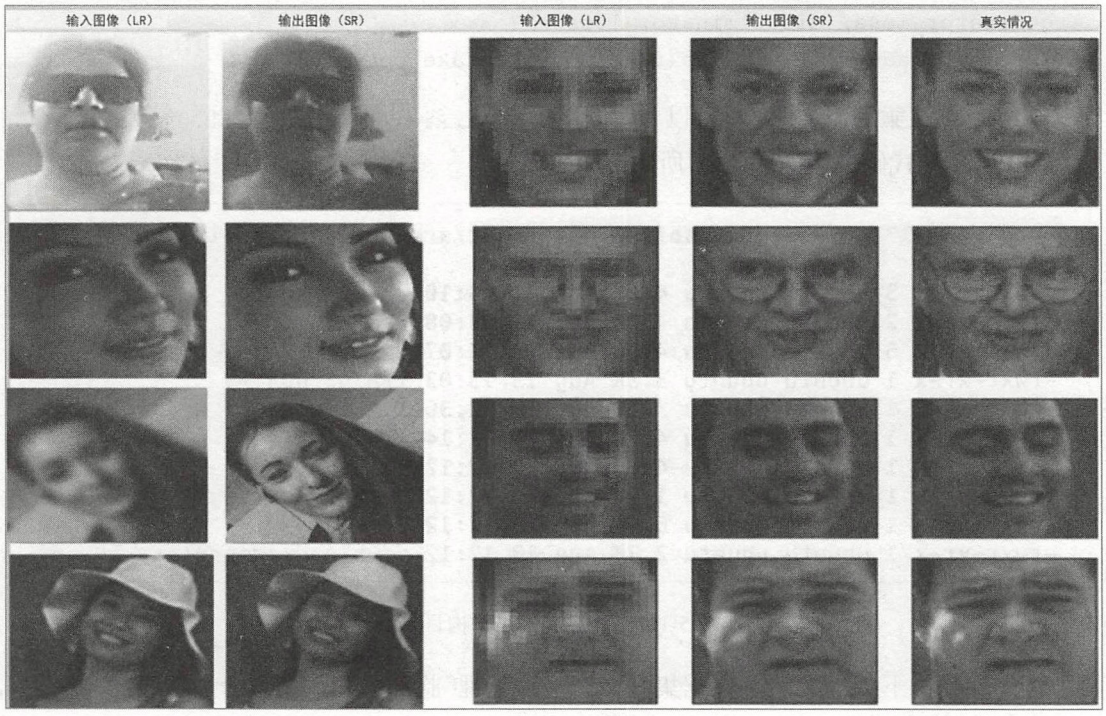


图5-13：生成的脸部超清晰图像示例

5.3 利用 DeepDream 生成梦幻的艺术图像

DeepDream 算法是一个经过修改的神经网络，可以用通过更改图像训练的方向来生成超现实主义、梦幻风格的图像。在网络的传输过程中 DeepDream 利用反向传播来修改图像而不是权重。

大致来讲，DeepDream 算法可以被总结为以下几步：

1. 选择一个你感兴趣的网络层和过滤器。
2. 然后从第一层开始计算，一直到这一层的激发。
3. 将过滤器的激发反向传播给输入图像。
4. 将梯度与学习速率相乘，并将其添加到输入图像。
5. 重复步骤 2 到步骤 4，直到得到你满意的结果。

在输出上迭代应用该算法，并且在每次迭代之后应用一些缩放使得网络可以通过探索它所知道的一组事物来生成无穷尽的新印象。

下面让我们深入代码来生成梦幻的图像。我们将会在 Keras 中把以下设置应用于预训练的 VGG16 模型。请注意，除了使用预处理模型，我们也可以把这组设置用于你所选择的全新神经网络模型：

```
settings_preset = {
    'dreamy': {
        'features': {
            'block5_conv1': 0.05,
            'block5_conv2': 0.02
        },
        'continuity': 0.1,
        'dream_l2': 0.02,
        'jitter': 0
    }
}

settings = settings_preset['dreamy']
```

下面这个工具函数可以对图像进行基本的加载、大小调整，并将图像格式化成合适的张量格式：

```
def preprocess_image(image_path):
```

```
    img = load_img(image_path, target_size=(img_height, img_width))
    img = img_to_array(img)
    img = np.expand_dims(img, axis=0)
    img = vgg16.preprocess_input(img)
    return img
```

接下来我们计算连续损失以使图像具有连续一致性，以避免图像变得杂乱和模糊，这看起来像如下网址的论文中提到的全局变异损失（http://www.robots.ox.ac.uk/~vedaldi/assets/pubs/mahendran15understanding.pdf）：

```
def continuity_loss(x):
    assert K.ndim(x) == 4
    a = K.square(x[:, :img_height-1, :img_width-1, :] -
                 x[:, 1:, :img_width-1, :])
    b = K.square(x[:, :img_height-1, :img_width-1, :] -
                 x[:, :img_height-1, 1:, :])

# （a + b）是平方的空间梯度，1.25是超参数，之前的论文中曾提及超参数应该大于1
    return K.sum(K.pow(a+b, 1.25))
```

下面我们载入 VGG16 模型和预训练权重：

```
model = vgg16.VGG16(input_tensor=dream, weights='imagenet', include_top=False)
```

之后我们将特征层的 l2 范数加入损失中，再加上连续损失来保证图像的局部连续性，然后再一次给损失加上 l2 范数以避免像素采用非常高的值，随后根据损失来计算梦幻梯度：

```
loss += settings['continuity'] * continuity_loss(dream) / np.prod(img_size)
loss += settings['dream_l2'] * K.sum(K.square(dream)) / np.prod(img_size)
grads = K.gradients(loss, dream)
```

最后我们给输入图像加上一个随机抖动（random_jitter），并在生成图像的像素上运行 L-BFGS 优化器来最小化损失：

```
random_jitter = (settings['jitter']*2) * (np.random.random(img_size)-0.5)
x += random_jitter

# run L-BFGS
```

```
x, min_val, info = fmin_l_bfgs_b(evaluator.loss, x.flatten(),
                                 fprime=evaluator.grads, maxfun=7)
```

最终我们会对梦幻进行解码，并将其保存到生成图像文件中：

```
x = x.reshape(img_size)
x -= random_jitter
img = deprocess_image(np.copy(x))
fn = result_prefix + '_at_iteration_%d.png' % i
imsave(fn, img)
```

仅经过 5 次迭代就能生成下面这样梦幻般的图像（见图 5-14）。

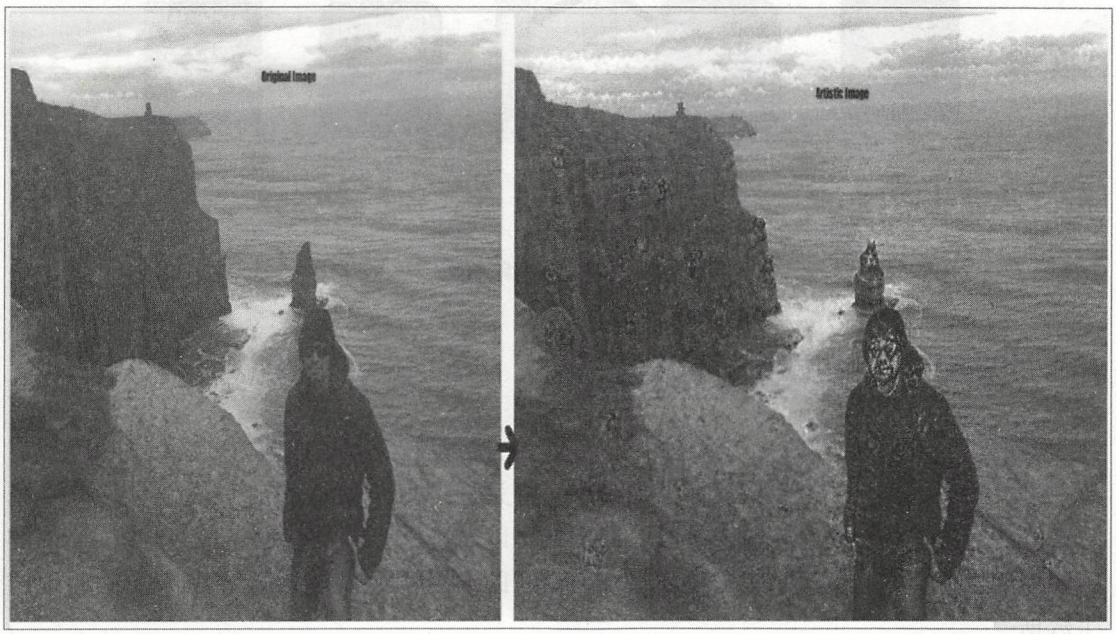


图5-14：左侧为原始图像，右侧为DeepDream生成的梦幻艺术图像

5.4 在 TensorFlow 上利用 VAE 生成手写数字

变分自动编码器（Variational Autoencoder，VAE）很好地将无监督学习和变分贝叶斯方法结合起来。它通过将输入、隐藏表征和重构输出作为有向图中的随机概率变量在基本的自动编码器上应用概率转换。

从贝叶斯的角度来看，编码器变成了一个变分推理网络，将观察到的输入映射到潜在空间的后验分布；解码器变成了一个生成网络，将任意的潜在坐标映射回原始数据空间的分布。

VAE 在生成潜在向量的编码网络上添加了约束，使得结果大致遵循了单位高斯分布（这个约束将 VAE 和标准的自动编码器分离），然后通过将潜在向量传输给解码器网络来重构图像（见图 5-15）。

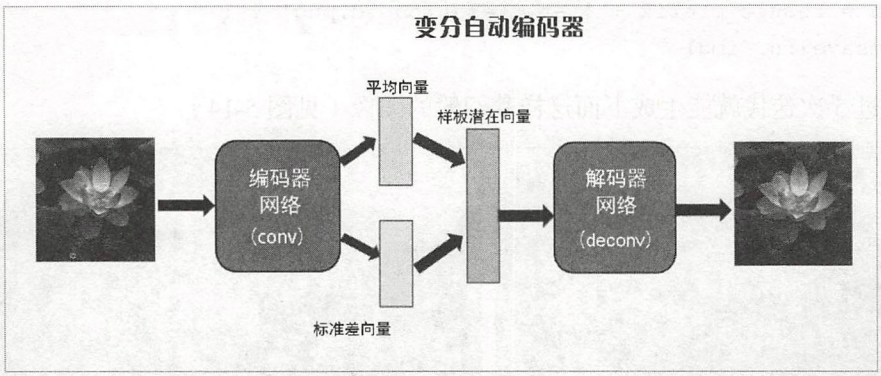


图5-15：变分自动编码器架构

5.5 VAE 在真实世界的比喻

假设我们要生成数据（一个动物），一个好的方法是在实际生成数据之前首先决定我们想要生成的数据类型。所以，我们必须设想一些关于代表动物的标准，比如它应该有 4 条腿并且能够游泳。一旦有了这些标准，我们就可以通过从动物界采样来生成动物。我们的想象标准类似于潜在变量。首先确定潜在变量有助于很好地描述数据，否则就是盲目地产生数据。

VAE 的基本思想是使用 $p(z|x)$来推断 $p(z)$。现在让我们展开介绍一些数学符号。

- x：表示我们想要生成的数据，这里是一个动物。
- z：表示潜在变量，这里是我们的想象。
- $p(x)$：表示数据的分布，这里是动物界。
- $p(z)$：表示潜在变量的正态概率分布，在这里是我们想象力的来源——大脑。
- $p(x|z)$：给定潜在变量生成数据的概率分布，在这里指的是将想象力变成真实的动物。

按照类似的例子，我们想把自己的想象力局限在动物界，所以我们不应该想象树根、树叶、钱、玻璃、GPU、冰箱、地毯等东西，因为这些东西不太可能与动物王国有什么共同之处。

VAE 的损失函数基本上是如下所示的一个包含正则化因子的负对数似然概率：

$$l_i(\theta,\varnothing) = -E_{Z\sim q_\theta(Z/x_i)}\left[\log p_\varnothing(x_i/z)\right] + KL\left(q_\theta(z/x_i)\middle\| p(z)\right)$$

第一项是预期的负对数似然或重构损失第 i 个数据点，其中，预期为根据编码器在表征上的分布计算所得。这一项有助于解码器很好地重构数据。如果不这样做，会产生巨大的成本。第二项代表编码器分布 $q(z|x)$和 $p(z)$之间的 Kullback-Leibler 散度，当编码器的输出表示 z 与正态分布不同时，它作为一个正则化因子来增加损失。

现在让我们深入代码，以便在 TensorFlow 上利用 MNIST 数据集和 VAE 来生成手写数字。首先我们制作一个隐藏层编码器网络 $Q(z|X)$，它以 X 为输入，以 $\mu(X)$为输出，并以 $\Sigma(X)$为高斯分布的一部分：

```
X = tf.placeholder(tf.float32, shape=[None, X_dim])
z = tf.placeholder(tf.float32, shape=[None, z_dim])

Q_W1 = tf.Variable(xavier_init([X_dim, h_dim]))
Q_b1 = tf.Variable(tf.zeros(shape=[h_dim]))

Q_W2_mu = tf.Variable(xavier_init([h_dim, z_dim]))
Q_b2_mu = tf.Variable(tf.zeros(shape=[z_dim]))
Q_W2_sigma = tf.Variable(xavier_init([h_dim, z_dim]))
Q_b2_sigma = tf.Variable(tf.zeros(shape=[z_dim]))


def Q(X):
    h = tf.nn.relu(tf.matmul(X, Q_W1) + Q_b1)
    z_mu = tf.matmul(h, Q_W2_mu) + Q_b2_mu
    z_logvar = tf.matmul(h, Q_W2_sigma) + Q_b2_sigma
    return z_mu, z_logvar
```

现在我们制作解码器网络 $P(X|z)$：

```
P_W1 = tf.Variable(xavier_init([z_dim, h_dim]))
```

```
P_b1 = tf.Variable(tf.zeros(shape=[h_dim]))

P_W2 = tf.Variable(xavier_init([h_dim, X_dim]))
P_b2 = tf.Variable(tf.zeros(shape=[X_dim]))

def P(z):
    h = tf.nn.relu(tf.matmul(z, P_W1) + P_b1)
    logits = tf.matmul(h, P_W2) + P_b2
    prob = tf.nn.sigmoid(logits)
    return prob, logits
```

接下来我们计算重构损失和 Kullback-Leibler 散度损失，并将它们相加，得出总共的 VAE 网络损失：

```
recon_loss = tf.reduce_sum(tf.nn.sigmoid_cross_entropy_with_logits
(logits=logits, labels=X), 1)
kl_loss = 0.5 * tf.reduce_sum(tf.exp(z_logvar) + z_mu**2 - 1. - z_logvar, 1)
# VAE 损失
vae_loss = tf.reduce_mean(recon_loss + kl_loss)
```

之后使用一个 `AdamOptimizer` 来最小化损失：

```
solver = tf.train.AdamOptimizer().minimize(vae_loss)
```

运行文件（`VAE.py` 或 `VAE.ipynb`）以开始在 `MNIST` 数据集上的 VAE 操作，生成图像会被保存到输出目录。在 10 000 次迭代后生成的手写数字图像样本如图 5-16 所示。

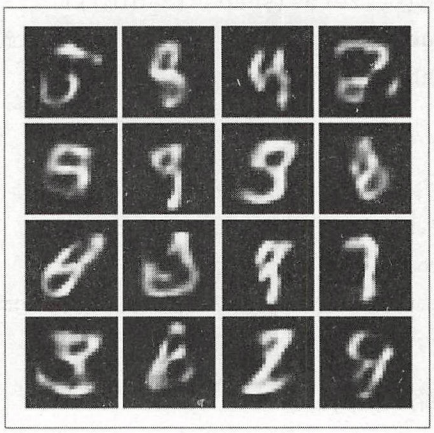


图5-16：VAE训练生成的数字图像

5.6 GAN 和 VAE 两个生成模型的比较

尽管两者都是非常令人兴奋的生成模型方法，并且均通过各自的生成能力帮助研究人员进入无监督领域，但这两种模型在训练方式上有所不同。GAN 根植于博弈论中，目的是找到两者达到纳什均衡的判别器网络和生成器网络。而 VAE 基本上是一个根植于贝叶斯推理的概率图模型，其目标是潜在地建模，即试图对基础数据的概率分布进行建模，以便从该分布中采样新的数据。

与 GAN 相比，VAE 有一个明确的已知评估模型质量的方法（如对数似然度，通过重要性采样或者下界估计），但是 VAE 的问题是，在计算潜在损失中使用直接均方误差而不是对抗网络，因为其局限在一个潜在的空间里工作，所以其过于简化了客观的任务，结果往往会产生比 GAN 更为模糊的图像。

5.7 总结

迁移学习有效地解决了深度学习在小数据集情况下的问题，并且无须重新制造轮子。你已经学会了从预训练模型进行特征提取和转换，并将其应用到自己的问题域。同时你也掌握了利用 Spark 和相关组件在大规模分布式系统上运行深度模型的方法。接下来你已经通过 SRGAN 中迁移学习的力量生成了逼真的高清晰度图像。你也掌握了 VAE 和 DeepDream 等其他生成模型的概念来生成艺术图像。在第 6 章中我们将会把关注点从训练深度模型和生成模型转换到利用多种方式在生产环境部署基于深度学习的应用。

6 将机器学习带入生产环境

大部分机器学习和深度学习相关的教程、教科书和视频主要关注在如何训练和评估模型。但是你应该如何才能在生产环境中训练模型，应用于实时的场景并给你的客户提供服务呢?

在本章中我们会利用 LFW 数据集开发一个脸部图像矫正系统，通过训练 GAN 模型来自动修复损坏的人脸图像。之后你将会学习分别在数据中心和云上基于容器的微服务环境中部署生产级别机器学习或者深度模型的方法。最后你将会学习在托管的云服务上使用无服务器环境部署深度模型。

本章将会包含以下内容：

- 利用 DCGAN 构建图像矫正系统。
- 部署机器学习模型的挑战。
- 微服务架构和容器。
- 多种部署深度学习模型的方法。
- 利用 Docker 部署基于 Keras 的深度学习服务。
- 在 GKE 云环境中部署深度模型。
- 基于 AWS Lambda 和 Polly 的无服务器图像与音频识别。
- 在托管云服务中进行人脸识别。

6.1 利用 DCGAN 构建一个图像矫正系统

图像矫正和修补是关于补充图像上丢失或者损坏部分的技术。构建一个通用的图像矫

正系统需要以下两方面的信息。

- **上下文信息**：根据丢失像素周边的像素信息进行推导。
- **感知信息**：根据现实生活或者其他图像来让修补的部分变得看起来更自然。

在本例中我们会利用 DCGAN 和**标记人脸图像**（Labeled Face in the Wild，LFW）来开发图像矫正系统。其中的 DCGAN 架构可以参考第 2 章。

在我们深入构建图像矫正系统之前先来定义一些术语和损失函数。

- $\boldsymbol{x}$：损坏的图像。
- $\boldsymbol{M}$：表示一个二进制掩码，其值为 1（表示这部分图像要被保留）或 0（表示这部分图像需要矫正）。$\boldsymbol{M}\odot\boldsymbol{x}$ 表示两个矩阵 $\boldsymbol{x}$ 和 $\boldsymbol{M}$ 之间的元素乘法并返回原始部分图像。
- p_{data}：样本数据的未知分布。

当我们训练好 DCGAN 中的判别器 $D(\boldsymbol{x})$和生成器 $G(\boldsymbol{z})$后，可以通过在丢失像素上最大化 $D(\boldsymbol{x})$来填补图像 $\boldsymbol{x}$ 中缺失的像素。

上下文损失在 $G(\boldsymbol{z})$没有根据输入图像周围像素生成近似图像时做出惩罚，惩罚值为 $\boldsymbol{x}$ 和 $G(\boldsymbol{z})$元素减法的差值：

$$\text{contextual}(\boldsymbol{z}) = \left\| \boldsymbol{M}\odot G(\boldsymbol{z}) - \boldsymbol{M}\odot\boldsymbol{x} \right\|$$

感知损失和最原始的 DCGAN 损失有着相同的标准，以确保修复的图像看起来更真实：

$$\text{perceptual}(\boldsymbol{z}) = \log\left(1 - D\left(G(\boldsymbol{z})\right)\right)$$

接下来我们需要在生成器 $G(\boldsymbol{z})$的结果中找到一个合理的结果图像，并将像素$(1-\boldsymbol{M})\odot G(\boldsymbol{z})$叠加到原始图像中，以获得最终的重构图像：

$$\text{xreconstructed} = \boldsymbol{M}\odot\boldsymbol{x} + (1-\boldsymbol{M})\odot G(\boldsymbol{z})$$

在 CPU 上训练深度卷积网络会十分缓慢，因此我们推荐你使用一个开启 CUDA 的 GPU 来进行深度学习相关的工作，包括对图像进行卷积和反卷积。

6.1.1 构建图像矫正系统的步骤

首先确认你已经下载了本章相关的代码：

1. DCGAN-ImageCorrection 项目会包含以下目录结构（见图 6-1）。

```
ubuntu@ip-172-31-6-47:~/software/kuntalg/DCGAN-ImageCorrection$ ls -lrth
total 544K
-rwxr-xr-x    1 ubuntu ubuntu 1.2K Aug  9 19:57 utils.py
-rwxr-xr-x    1 ubuntu ubuntu 7.8K Aug  9 19:57 dcgan.py
drwxr-xr-x    2 ubuntu ubuntu 4.0K Aug 27 15:30 complete
drwxr-xr-x    2 ubuntu ubuntu 4.0K Aug 27 15:36 complete_src
drwxrwxr-x 5751 ubuntu ubuntu 200K Aug 31 13:56 lfw
-rwxr-xr-x    1 ubuntu ubuntu 1.2K Aug 31 13:57 create_tfrecords.py
drwxr-xr-x    2 ubuntu ubuntu 288K Aug 31 13:58 data
-rwxrwxr-x    1 ubuntu ubuntu   15 Aug 31 16:02 test.sh
drwxr-xr-x    2 ubuntu ubuntu 4.0K Sep 10 13:53 checkpoints
-rwxrwxr-x    1 ubuntu ubuntu 6.4K Sep 10 13:54 train_generate.py
-rwxrwxr-x    1 ubuntu ubuntu 5.8K Sep 10 13:55 image_correction.py
```

图6-1：DCGAN-ImageCorrection代码目录结构

2. 从 http://vis-www.cs.umass.edu/lfw 下载 LFW 数据集，并将其解压缩到 lfw 目录。

```
wget http://vis-www.cs.umass.edu/lfw/lfw-funneled.tgz
tar -xvzf lfw-funneled.tgz
```

3. 运行 create_tfrecords.py 将 LFW 图像转换成 TensorFlow 标准格式。在 Python 文件中修改你的 LFW 图像目录位置。

```
base_path = <Path to lfw directory>
python create_tfrecords.py
```

这会在你的 data 目录下产生下面的 tfrecords 文件（见图 6-2）。

```
ubuntu@ip-172-31-6-47:~/software/kuntalg/DCGAN-ImageCorrection$ ls data
Aaron_Eckhart.tfrecords
Aaron_Guiel.tfrecords
Aaron_Patterson.tfrecords
Aaron_Peirsol.tfrecords
Aaron_Pena.tfrecords
Aaron_Sorkin.tfrecords
Aaron_Tippin.tfrecords
Abba_Eban.tfrecords
Abbas_Kiarostami.tfrecords
Abdel_Aziz_Al-Hakim.tfrecords
Abdel_Madi_Shabneh.tfrecords
```

图6-2：tfrecords文件

4. 现在利用下面的命令训练 DCGAN 模型。

```
python train_generate.py
```

你可以通过修改 Python 文件中的 `max_itr` 属性来调整训练的最大迭代次数。训练开始后每经过 5000 次迭代，你就可以在 `lfw-gen` 目录看到如图 6-3 所示的生成图像。

图6-3：生成人脸图像示例

5. 最后你可以利用训练好的 DCGAN 模型来修复损坏的图像。你需要将损坏的图像放在 `complete_src` 目录并执行下面的命令。

```
python image_correction.py --is_complete True --latest_ckpt
<checkpoint number>
```

你也可以通过调整 `center` 和 `random` 以及 `masktype` 属性来调整前面命令的掩码类型（见图 6-4）。

```
ubuntu@ip-172-31-6-47:~/software/kuntalg/DCGAN-ImageCorrection$ python image_correction.py --is_complet
e True --latest_ckpt 20000
2017-09-16 14:13:32.984245: W tensorflow/core/platform/cpu_feature_guard.cc:45] The TensorFlow library
wasn't compiled to use SSE4.1 instructions, but these are available on your machine and could speed up
CPU computations.
2017-09-16 14:13:32.984376: W tensorflow/core/platform/cpu_feature_guard.cc:45] The TensorFlow library
wasn't compiled to use SSE4.2 instructions, but these are available on your machine and could speed up
CPU computations.
2017-09-16 14:13:32.984400: W tensorflow/core/platform/cpu_feature_guard.cc:45] The TensorFlow library
wasn't compiled to use AVX instructions, but these are available on your machine and could speed up CPU
 computations.
Restoring variables:
 g/reshape/dense/kernel:0
 g/reshape/dense/bias:0
 g/reshape/batch_normalization/beta:0
 g/reshape/batch_normalization/gamma:0
 g/deconv1/conv2d_transpose/kernel:0
```

图6-4：启动模型训练

前面的命令会在 complete 目录中生成如图 6-5 所示的修复图像。

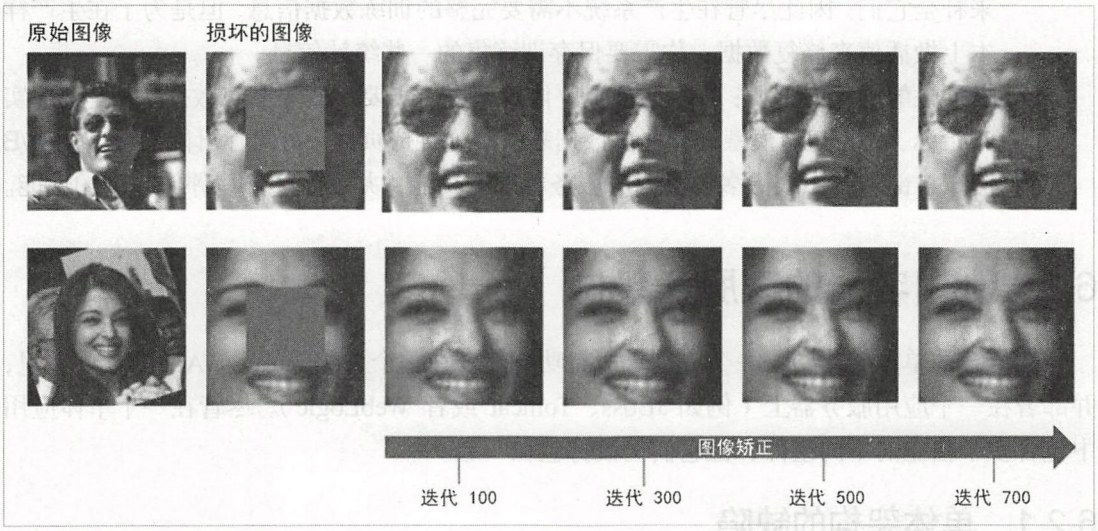


图6-5：图像修复示例

6.1.2 在生产环境部署模型的挑战

大部分研究者和机器学习从业者主要关注机器学习或者深度学习模型的训练和评估。研究者构建模型的过程就好像在自己家里做饭，而在生产环境构建和部署模型就像在饭店里给更多不同的顾客（口味还经常变化）做饭。

在生产环境部署模型的一些常见挑战如下。

- **扩展性：**一个真实世界的生产环境和一个训练或者研究环境会十分不同。你通常需要考虑在大流量下处理数据对性能的影响。你的模型需要能自动地根据流量情况进行扩容，并在流量下降时进行缩容。
- **模型训练和更新的自动化：**真实世界的数据是时时刻刻在变化的，因此当你的模型进入真实世界的生产环境时，所看到的数据也会和之前训练时看到的不同。这就意味着你需要重新训练自己的模型（有时需要自动化），并在多个模型之间无缝切换。
- **开发语言之间的交互：**通常会有两组不同的人分别负责模型的研究（训练）和将其应用的生产环境，因此研究时用到的语言和生产环境使用的语言通常也会不同。尽管模型在本质上是相同的，但是不同语言在实现机器学习模型上会有差异，这导致了很多问题。

- **训练集元数据的知识：**真实生产的数据通常会缺少某些值，你需要应用插值的方法来补全它们。因此尽管在生产系统不需要完整的训练数据信息，但是为了在生产样本上做插值来修复数据，你需要保存训练集的一些统计信息。
- **模型性能的实时评估：**在生产系统评估模型性能需要你收集真实数据（或者其他实时指标），并随着模型处理更多数据来动态地生成页面。有时你可能需要应用 A/B 测试，部署两个或更多模型同时服务相同的功能，并在生产环境评估它们的性能。

6.2 利用容器的微服务架构

在传统的单体架构中，一个应用将所有功能整合在一个 EAR 或者 WAR 的软件包里，并部署在一个应用服务器上（例如 JBoss、Tomcat 或者 WebLogic）。尽管在一个单体应用中包含多个独立的不同组件，但它们仍被打包在一起。

6.2.1 单体架构的缺陷

单体架构在设计上的缺陷有如下几个：

- 单体架构中的各个功能组件被打包在一起，没有进行隔离。因此，更改其中的一个组件就需要更新整个应用并导致应用的停止，这对于生产环境的场景是不可接受的。
- 对单体应用的扩容，通常效率会很低，因为扩容过程中你需要在多个不同的服务器上部署应用（WAR 或者 EAR），并在每台服务器上消耗相同的资源。
- 在真实世界中通常只有一两个功能组件被频繁使用，但是在单体的设计下所有组件使用相同的资源，因此很难将频繁使用的组件分离出来提升应用的整体性能。

微服务（Microservice）是一个将大型软件项目拆分成通过简单 API 进行通信的多个松耦合组件的技术。基于微服务的架构将每个功能设计成单独的服务来避免在单体设计中的缺陷。

6.2.2 微服务架构的优点

微服务设计模式的优点有如下几个。

- **单一责任原则：**微服务架构确保每个功能均分别部署并通过简单的 API 对外提供服务。

- **高扩展性**：使用频繁或按需使用的服务可以部署在多个服务器上来支撑高请求和流量以提升性能。这对于单体服务来说是很困难的。
- **提升容错能力**：单独一个模块的失败不会影响整个应用，并且由于模块以单独服务的形式运行，因此你可以重启失败模块进行快速的恢复。然而，在单体服务中一个模块的错误可能会影响到其他模块。
- **技术栈的自由**：微服务架构允许你选择对某个模块最为合适的技术，并可以帮助你在单独的服务上尝试新的技术栈。

部署微服务应用的最佳方式是通过容器。

1. 容器

容器是通过在宿主机上共享操作系统的内核（二进制文件和库）来实现的轻量级共享进程。容器通过一组抽象同时解决了很多复杂的问题。容器的流行可以用下面几个词来描述：隔离性！可移植性！重复性！

2. Docker

Docker 是当下最为火热的开源项目，也是一个十分流行的容器引擎。它可以帮助你以一种简单的方法将应用及其依赖进行打包，以在本地或云端进行部署。

3. Kubernetes

Kubernetes 是一个 Google 的开源项目，它提供了容器的编排，包括自动扩容、服务发现、负载均衡等多个功能。简单来说，它可以自动化云上容器化应用管理。

在本章中我们提到的容器引擎都是 Docker，但是还存在着其他的容器提供类似的特点或功能。

6.2.3 使用容器的优点

使用容器的优点有下面几个。

- **持续部署和测试**：通常一次发布的生命周期包含了多个不同的环境，例如生产和开发环境可能会有不同的包版本或依赖。Docker 通过在生产环境和开发环境保持同样的配置和依赖弥补了两者间的距离，保证环境的一致性。因此，你可以在开发和生产环境中使用同样的容器而无须任何手动的修改。

- **多云平台支持**：Docker 最大的一个优点是它可在多个环境和平台之间进行移植。所有的主流云提供商例如 Amazon Web Service（AWS）和 Google Compute Platform（GCP）都对 Docker 提供了单独的支持（AWS ECS 或 Google GKE）。Docker 容器也可以运行在任何支持 Docker 的操作系统的虚拟机上（Amazon EC2 或 Google Compute Engine）。
- **版本控制**：Docker 容器可以像版本控制系统 `Git/SVN` 一样工作，你可以给 Docker 镜像做一个变更，并进行版本控制。
- **隔离性和安全性**：Docker 确保运行在容器中的应用和其他部分是完全隔离的，并提供完整的流量管控和资源管理。Docker 容器中运行的进程不能访问其他容器中的进程。从架构的角度来看，每个容器拥有自己独立的一组资源。

你可以将机器学习和深度学习应用与 Docker 的部署能力相结合，使得系统更加有效、可共享。

6.3 部署深度模型的多种方法

机器学习让人兴奋并且有趣。但是，如果想让你的模型服务于真实的人和系统，则在模型的训练和部署方面都有很多挑战。

在生产环境部署机器学习模型有多种方法，在选择这些不同的方法时你需要考虑下面几个因素：

- 你希望模型是实时流分析系统的一部分还是批处理分析系统的一部分？
- 你希望有多个模型处理相同的功能还是需要对模型进行 A/B 测试？
- 你希望模型的更新频率是怎样的？
- 如何根据流量进行模型的扩展？
- 如何和其他服务集成，或者如何集成到已有的机器学习流水线？

6.3.1 方法 1——离线建模和基于微服务的容器化部署

在这种方法下，你会离线训练和评估模型，利用训练好的模型构建 RESTful 服务并通过容器部署。接下来你会根据成本、安全性、扩容以及基础设施的需求选择是在数据中心还是云上运行容器。这种方法适合于你的机器学习服务有着持续的流量访问，并且需要根据流量的波动进行服务的动态扩展。

6.3.2 方法 2——离线建模和无服务器部署

在这种方法下，你将离线训练模型并在无服务器环境部署你的服务，例如 AWS Lambda（你无须以小时/分钟为计费单位对容器或虚拟机进行付费，只需要为 API 调用次数进行付费）。这种方法适合于你的模型服务并不会被持续访问而是在某个固定的时间段被调用。不过即使有持续的流量访问（根据请求的数量），这种方法和第一种方法相比依然可能更节约成本。

6.3.3 方法 3——在线学习

有时你需要通过将机器学习服务和现有流水线结合进行实时数据流分析（例如在客户端的消息队列处理 IOT 传感器数据）。数据在实时数据流场景下的变化会十分频繁。在这种场景下，离线模型训练不是一个好的选择。取而代之的是，你需要自动化地根据看到的数据对模型进行调整——利用 SGD 或其小批变种来调整模型的权重和参数。

6.3.4 方法 4——利用托管机器学习服务

这种方法适用于当你没有足够的资源和团队来构建自己的机器学习模型时。你可以利用已有的基于云的机器学习和深度学习服务，例如 Google Cloud ML、Azure ML、AWS Rekognition、AWS Polly、Google Cloud Vision 等，通过简单地调用 API 来满足你的需求。

接下来，我们会通过例子来展示上面提到的几种部署方式。

6.4 在 Docker 上运行基于 Keras 的深度模型

在本例中，我们通过预训练的 Keras InceptionV3 模型构建一个图像识别系统，并用容器部署在本机上。参考第 4 章可获取更多关于预训练模型的信息。我们预训练的 Keras 模型会在 Docker 容器中运行并利用一个 Flask 实现的 REST API 来对外提供服务（见图 6-6）。

请确保你已经准备好 `keras-microservice` 项目并执行下面的步骤来在容器中运行一个基于 Keras 的深度模型。

1. 首先检查当前工作目录下的 Dockerfile 并构建 Docker 镜像（见图 6-6）：

```
docker build -t keras-recognition-service .
```

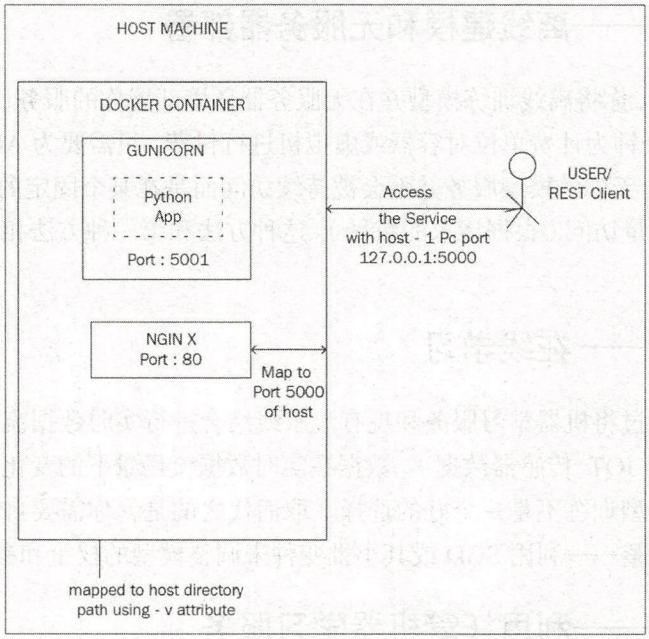


图6-6：基于Flask的REST服务架构

2. 当镜像构建成功后利用 `docker run` 命令来运行容器：

```
docker run -it --rm -d -p <host port>:<container port> -v <host
path>:<container path> keras-recognition-service
```

例如：

```
docker run -it --rm -d -p 5000:80 -v /Users/kuntalg/knowledge:/
deep/model keras-recognition-service
```

在Docker容器内，Keras模型运行在5001端口上的一个名为Gunicorn的Python WSGI HTTP服务器上，并通过Nginx反向代理到80端口。我们利用前面提到的-p参数来将容器端口映射到宿主机上。同时我们利用-v 将宿主机上的一个路径以卷的方式挂载到容器内的一个路径，这样就可以通过这个路径载入之前训练好的模型。

现在我们可以通过执行 `test.sh` 脚本来测试图像识别服务。这个脚本包含了一个 `curl` 命令来调用我们暴露的图像识别服务 REST API 并测试：

```
#!/bin/bash
echo "Prediction for 1st Image:"
echo "---------------------------------"
(echo -n '{"data": "'; base64 test-1.jpg; echo '"}') |
curl -X POST -H "Content-Type: application/json" -d @-
http://127.0.0.1:5000

echo "Prediction for 2nd Image:"
echo "---------------------------------"
(echo -n '{"data": "'; base64 test-1.jpg; echo '"}') |
curl -X POST -H "Content-Type: application/json" -d @-
http://127.0.0.1:5000
```

3. 最后我们执行下面的脚本来生成 Keras 服务的预测（见图 6-7）：

```
./test_requests.sh
```

```
a0999b1381a5:keras-microservice kuntalg$ ./test_requests.sh
Prediction for 1st Image:
---------------------------------
{
  "predictions": [
    {
      "description": "cheetah",
      "label": "n02130308",
      "probability": 61.97998523712158
    },
    {
      "description": "leopard",
      "label": "n02128385",
      "probability": 23.502658307552338
    },
    {
      "description": "jaguar",
      "label": "n02128925",
      "probability": 1.483460795134306
    }
  ]
}
Prediction for 2nd Image:
---------------------------------
{
  "predictions": [
    {
      "description": "macaw",
      "label": "n01818515",
      "probability": 69.18083429336548
    },
    {
      "description": "bee_eater",
      "label": "n01828970",
      "probability": 4.089190810918808
    },
    {
      "description": "lorikeet",
      "label": "n01820546",
      "probability": 1.0934238322079182
    }
  ]
}
```

图6-7：生成预测结果

我们已经成功地利用容器部署了第一个基于 Keras 的深度学习模型。

6.5 在 GKE 上部署深度模型

当我们已经训练好深度模型，并通过容器在本机部署并产生预测后，下一步就可以利用 Kubernetes 和 Docker 将模型部署到云上（在本例中为 Google Cloud）。

执行下面的步骤将一个本地容器化的模型带到云上。

1. 注册 Google Cloud 试用账号（`https://cloud.google.com/free`），在 New Project 界面中输入一个相关的 Project name 来新建项目（见图 6-8）。

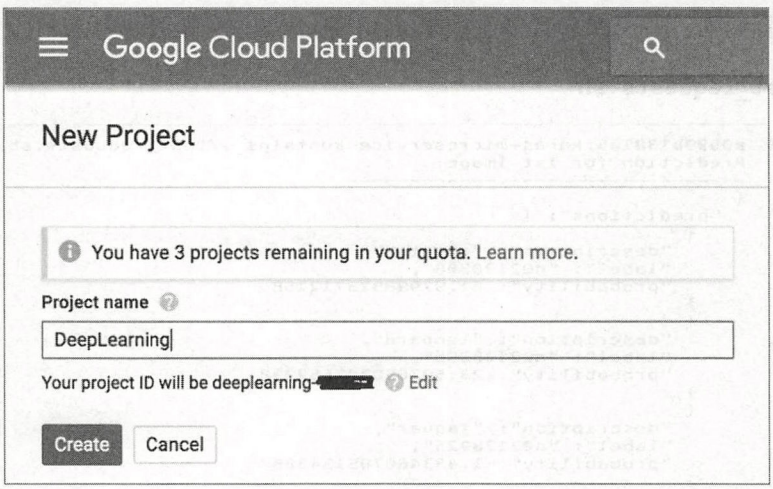


图6-8：新建项目

请注意下面的 project ID 包含了你的 Project name 以及其他一些格式为`<project name>-xxxxxx` 的数字。当我们在云端部署你本季度的模型时需要这个 project ID。

2. 在机器上安装 Google Cloud SDK（`https://cloud.google.com/sdk`），接下来安装 `kubectl` 来管理你的 Kubernetes 集群：

```
gcloud components install kubectl
```

`gcloud` 命令已经包含于 Google Cloud SDK 中。

3. 设置 `gcloud` 命令行工具 `config` 的环境变量：

```
gcloud config set project <project ID>
```

```
gcloud config set compute/zone <zone name such as us-central1-b>
export PROJECT_ID="$(gcloud config get-value project -q)
```

4. 构建一个带标签或者版本（本例中为 v1）的 Docker 镜像：

```
docker build -t gcr.io/<project ID>/keras-recognition-service:v1 .
```

例如：

```
docker build -t gcr.io/deeplearning-123456/keras-recognitionservice: v1 .
```

5. 通过 docker push 上传之前构建好的镜像到 Google Container Registry：

```
gcloud docker -- push gcr.io/<project ID>/keras-recognitionservice: v1
```

例如（见图 6-9）：

```
gcloud docker -- push gcr.io/deeplearning-123456/kerasrecognition-
service:v1
```

```
a0999b1381a5:keras-microservice kuntalg$ gcloud docker -- push gcr.io/deeplearning-[redacted]/keras-recognition-service:v1
The push refers to a repository [gcr.io/deeplearning-[redacted]/keras-recognition-service]
417e16762df2: Pushing [==================================================>]  12.29kB
e1780d965c92: Pushing [==================================================>]  3.584kB
23dd5f2636d7: Pushing [==================================================>]   2.56kB
a6325a4c40cd: Pushing   2.56kB
afb4aa79c500: Pushing [=>                                                ]  14.31MB/549.4MB
fa527337b328: Waiting
5582a9120302: Waiting
f25422bf99b3: Waiting
f74517661c76: Waiting
0566c118947e: Waiting
6f9cf951edf5: Waiting
182d2a55830d: Waiting
5a4c2c9a24fc: Waiting
cb11ba605400: Waiting
```

图6-9：上传镜像到镜像仓库

6. 当容器镜像存储到仓库之后，我们需要指定 Compute Engine 虚拟机实例的个数来创建一个容器集群。这个集群会通过 Kubernetes 进行管理和编排。执行下面的命令来创建一个名为 dl-cluster 的两节点集群：

```
gcloud container clusters create dl-cluster --num-nodes=2
```

7. 利用 Kubernetes 的 kubectl 命令行工具来在容器集群中部署并运行一个监听在 80 端口的容器（见图 6-10）：

```
gcloud container clusters get-credentials dl-cluster
```

```
kubectl run keras-recognition-service --image=gcr.io/
deeplearning-123456/keras-recognition-service:v1 --port 80
```

```
a0999b1381a5:keras-microservice kuntalg$ kubectl run keras-recognition-service --image=gcr.io/deeplearning-[redacted]/keras-recogni
tion-service:v1 --port 80
```

图6-10：利用kubectl启动服务

8. 将运行在容器集群中的应用绑定到一个负载均衡器上，这样我们就可以把图像识别服务暴露给真实世界的用户：

```
kubectl expose deployment keras-recognition-service
--type=LoadBalancer --port 80 --target-port 80
```

9. 接下来，运行下面的 kubectl 命令来获得服务的外部 IP：

```
kubectl get service
```

10. 最后执行下面的命令来获得在云端容器集群中图像识别服务的预测结果（见图6-11）：

```
(echo -n '{"data": "'; base64 test-1.jpeg; echo '"}') | curl -X POST -H
"Content-Type: application/json" -d @- http://<External IP>:80
```

```
a0999b1381a5:keras-microservice kuntalg$ (echo -n '{"data": "'; base64 test-1.jpg; echo '"}') | curl -X POST -H "Content-Type:
application/json" -d @- http://[redacted]:80
{
  "predictions": [
    {
      "description": "cheetah",
      "label": "n02130308",
      "probability": 61.979931592941284
    },
    {
      "description": "leopard",
      "label": "n02128385",
      "probability": 23.502694070339203
    },
    {
      "description": "jaguar",
      "label": "n02128925",
      "probability": 1.4834615401923656
    }
  ]
```

图6-11：访问图像识别服务

6.6 利用 AWS Lambda 和 Polly 进行无服务器的图像识别并生成音频

在本例中，我们将利用 TensorFlow 和预训练 InceptionV3 模型构建一个图像预测生成音频的系统并将其以无服务器的方式部署在 AWS Lambda 之上。我们会在 AWS Lambda 上运行图像预测，从 S3 载入预训练模型并通过 AWS API 网关对外暴露服务（见图 6-12）。

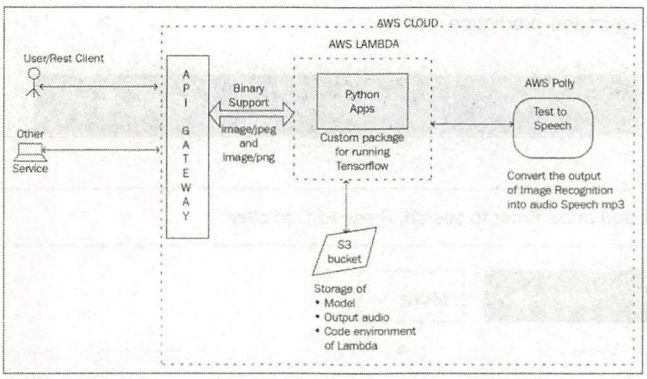


图6-12：基于AWS Lambda和API网关的架构

执行下面的步骤来利用无服务器平台构建基于音频的图像识别系统。

1. 注册一个 AWS 试用账户（`https://aws/amazon.com/free`），导航到 IAM 服务来创建一个新的 AWS Lambda 角色。绑定两个新的策略：S3FullAccess 和 PolluFullAccess 以及内置的 lambda_basic_execution 策略（见图 6-13）。

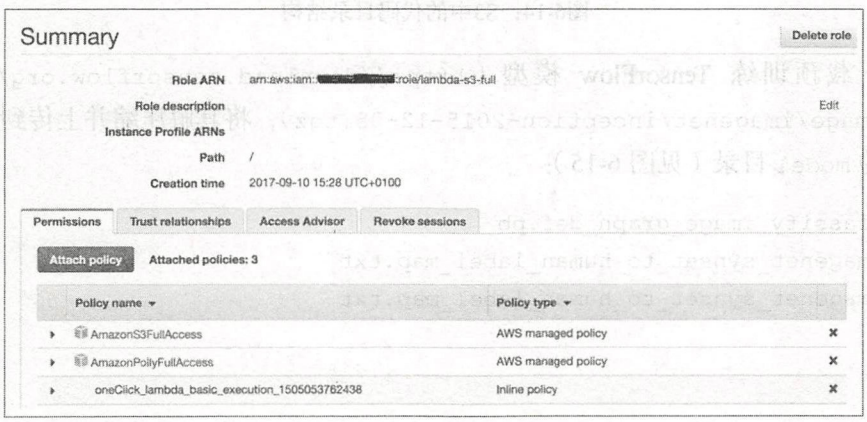


图6-13：创建IAM规则

2. 创建一个 S3 桶来存储 Lambda 代码（包括自定义的 Python 包，例如 `numpy`、`scipy`、`tensorflow` 等），同时在 S3 桶下创建 3 个目录：
   - `code`：将 Lambda 环境下的代码存放在这里。
   - `audio`：预测的音频会被存放在这里。
   - `model`：预训练模型存放在这里（见图 6-14）。

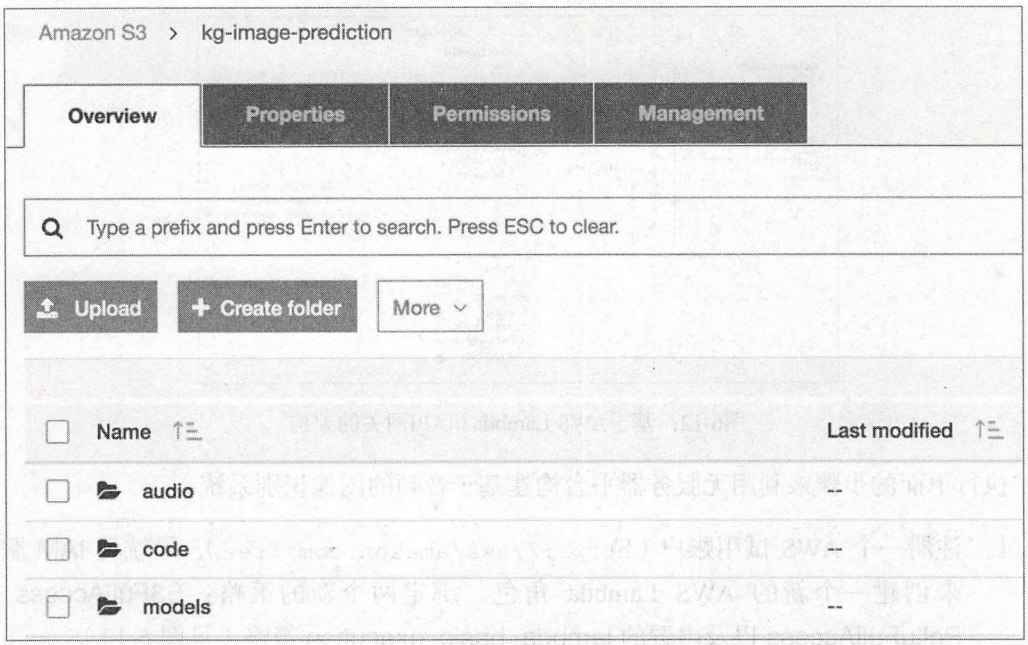


图6-14：S3中的代码目录结构

3. 下载预训练 TensorFlow 模型（`http://download.tensorflow.org/models/image/imagenet/inception-2015-12-05.tgz`），将其解压缩并上传到 S3 桶下的 `model` 目录（见图 6-15）：

```
classify_image_graph_def.pb
imagenet_synset_to_human_label_map.txt
imagenet_synset_to_human_label_map.txt
```

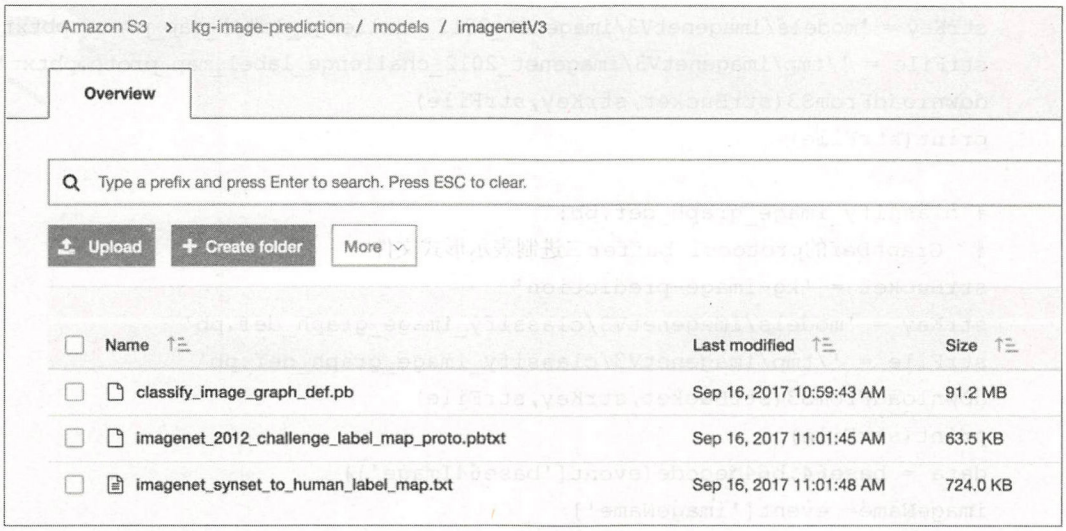


图6-15：S3中的预训练模型目录

4. lambda_tensorflow.zip压缩包中包含了一个classify.py文件，其会在Lambda函数中被执行。更改 classify.py 文件中桶的名字，再次压缩并上传至 S3 桶的 code 目录下（见图 6-16）：

```
def lambda_handler(event, context):

    if not os.path.exists('/tmp/imagenetV3/'):
        os.makedirs('/tmp/imagenetV3/')

    # imagenet_synset_to_human_label_map.txt:
    #   synset ID到人类可读字符串的映射文件
    strBucket = 'kg-image-prediction'
    strKey = 'models/imagenetV3/imagenet_synset_to_human_label_map.txt'
    strFile = '/tmp/imagenetV3/imagenet_synset_to_human_label_map.txt'
    downloadFromS3(strBucket,strKey,strFile)
    print(strFile)

    # imagenet_2012_challenge_label_map_proto.pbtxt:
    # 标签到synset ID映射的protocol buffer文本形式文件

    strBucket = 'kg-image-prediction'
```

```
strKey = 'models/imagenetV3/imagenet_2012_challenge_label_map_proto.pbtxt'
strFile = '/tmp/imagenetV3/imagenet_2012_challenge_label_map_proto.pbtxt'
downloadFromS3(strBucket,strKey,strFile)
print(strFile)

# classify_image_graph_def.pb:
#  GraphDef的protocol buffer二进制表示形式文件
strBucket = 'kg-image-prediction'
strKey = 'models/imagenetV3/classify_image_graph_def.pb'
strFile = '/tmp/imagenetV3/classify_image_graph_def.pb'
downloadFromS3(strBucket,strKey,strFile)
print(strFile)
data = base64.b64decode(event['base64Image'])
imageName= event['imageName']

image=io.BytesIO(data)
strBucket = 'kg-image-prediction'

strKey = 'raw-image/tensorflow/'+imageName+'.png'
uploadToS3(image, strBucket, strKey)
print("Image file uploaded to S3")


audioKey=imageName+'.mp3'
print(audioKey)
print("Ready to Run inference")

strBucket = 'kg-image-prediction'
strKey = 'raw-image/tensorflow/'+imageName+'.png'
imageFile = '/tmp/'+imageName+'.png'
downloadFromS3(strBucket,strKey,imageFile)
print("Image downloaded from S3")

strResult = run_inference_on_image(imageFile)

# 调用AWS Polly从文本生成语音
polly_client=boto3.client('polly')
```

```
    response = polly_client.synthesize_speech(Text =strResult,OutputFormat
= "mp3",VoiceId = "Joanna")
    if "AudioStream" in response:
        output = os.path.join("/tmp", audioKey)
        with open(output, "wb") as file:
            file.write(response["AudioStream"].read())

    # 上传语音到S3
    print("Ready upload to S3 audio")
    strBucket = 'kg-image-prediction'
    strKey = 'audio/'+audioKey
    strFile = '/tmp/'+audioKey

    with open(strFile, 'rb') as data:
        uploadToS3(data,strBucket,strKey)
    # 清理当前目录
    os.remove(imageFile)
    os.remove(strFile)

    return strResult
```

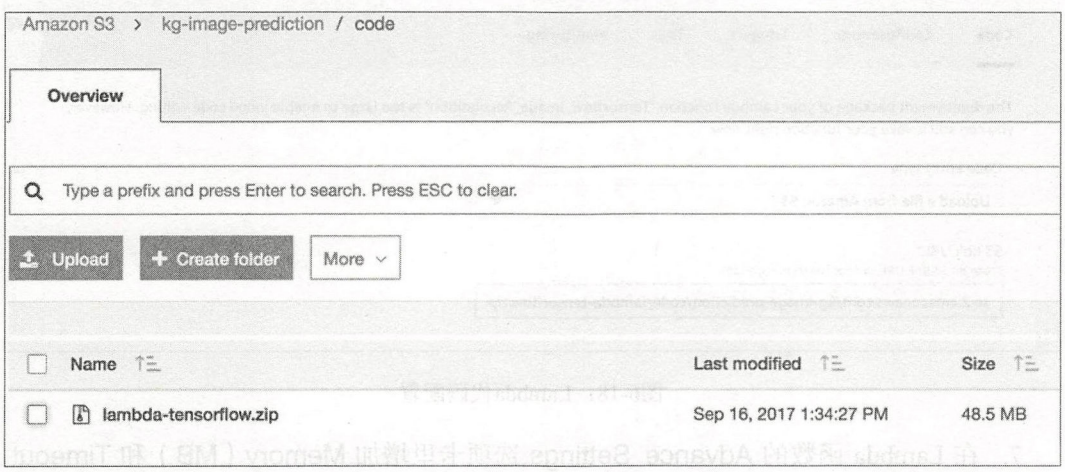


图6-16：上传压缩代码到S3

5. 现在通过 Web 控制台导航至 Lambda 服务，从零开始创建一个新的 Lambda 函数。填写函数的**名称**和**描述**，选择 Python 2.7 作为**运行环境**，并绑定之前创建的 IAM 角色给这个 Lambda 函数（见图 6-17）。

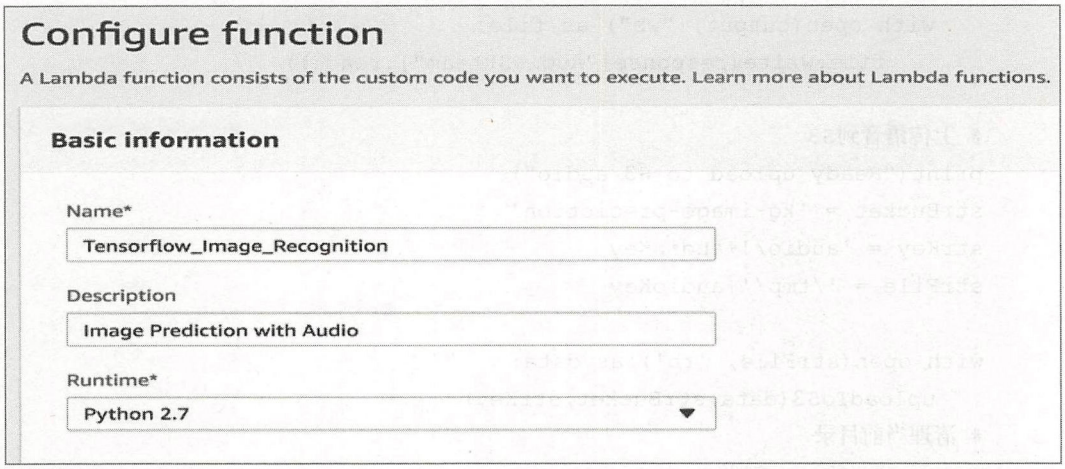


图6-17：Lambda基本信息配置

6. 在 Lambda 函数配置里指定代码（`lambda_tensorflow.zip`）的位置（见图 6-18）：

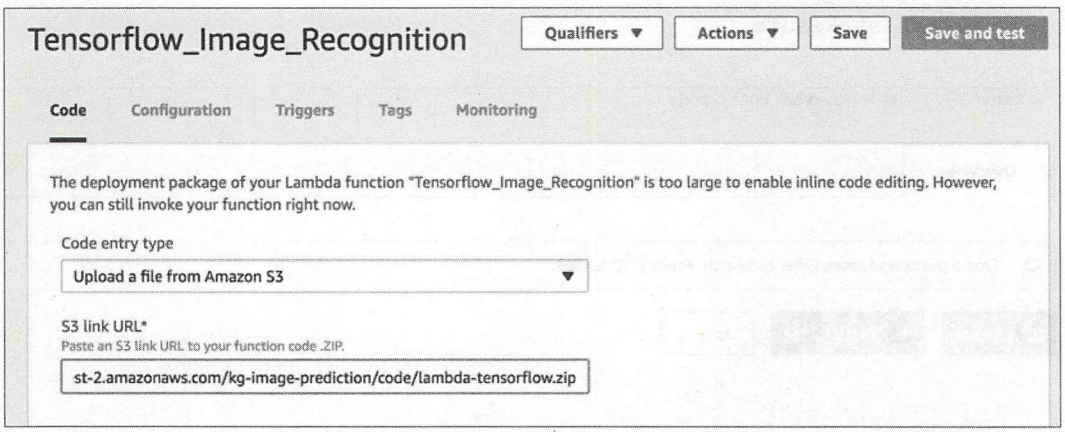


图6-18：Lambda代码配置

7. 在 Lambda 函数的 Advance Settings 选项卡里增加 Memory（MB）和 Timeout。首次运行 Lambda 时由于需要从 S3 加载预训练模型，因此将会花费一些时间（见图 6-19）。

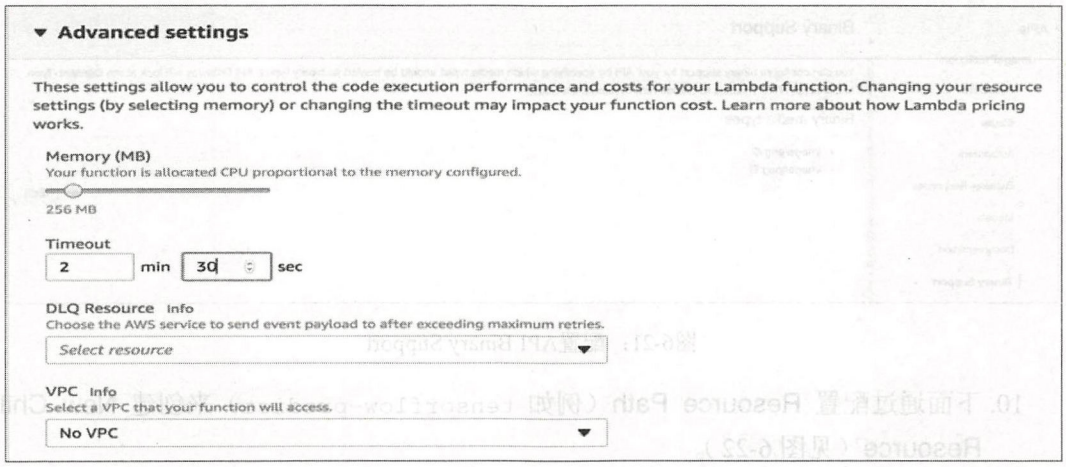


图6-19：Lambda高级配置

8. 导航至 API Gateway 服务来创建一个新的 API（见图 6-20）。

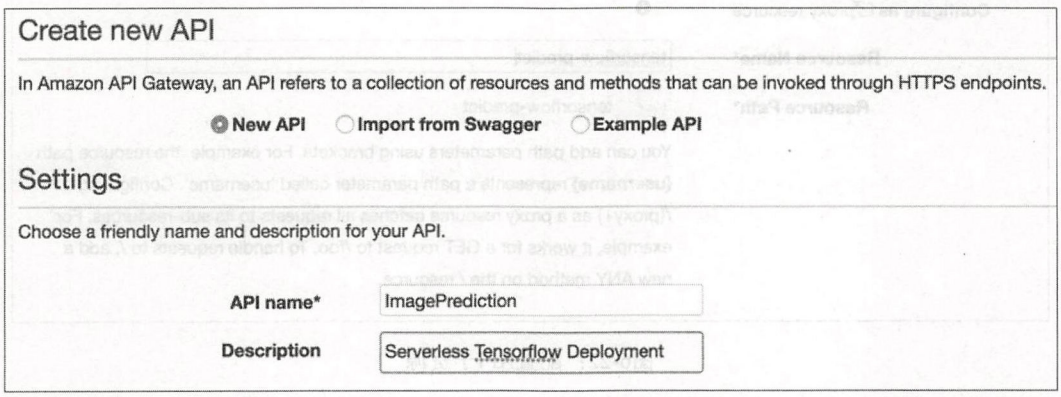


图6-20：新建API

9. 单击左侧目录的 Binary Support 选项卡来增加下面两种内容类型（见图 6-21）：

- image/png
- image/jpeg

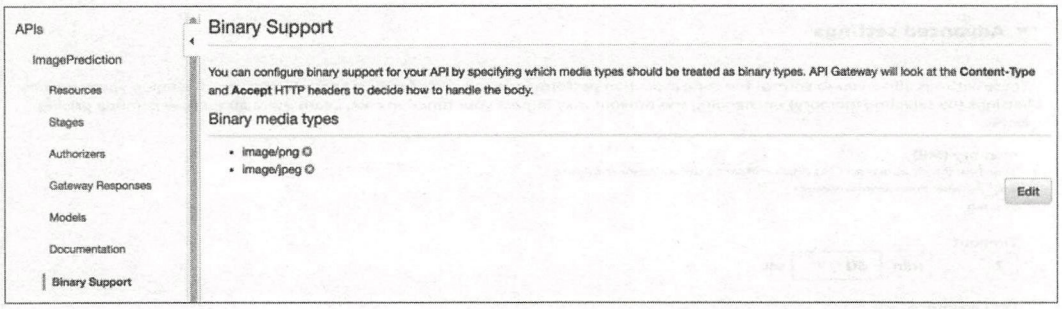


图6-21：配置API Binary Support

10. 下面通过配置 Resource Path（例如 `tensorflow-predict`）来创建 New Child Resource（见图 6-22）。

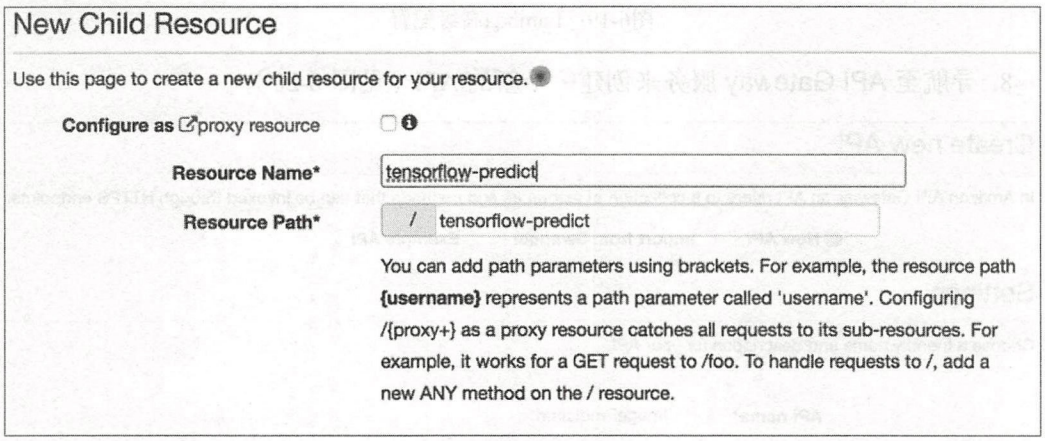


图6-22：配置API子资源

11. 之后通过单击 Action 菜单中的 Create Method 给子资源增加方法（POST）。将 Lambda 函数以及 API 资源加入 AMP 中。在这里你需要从下拉列表中选择 Lambda 函数的正确区域。
12. 当 POST 方法被创建时，单击 Integration Request 并展开 Body Mapping Templates 选项卡。在 Request body passthrough 里选择 When there are no templates defined (recommended)。接下来在 Content-Type 里增加 image/jpeg，并在 Generate template 里增加下面的内容（见图 6-23）：

```
{
"base64Image": "$input.body",
```

```
"imageName": "$input.params(imageName)"
}
```

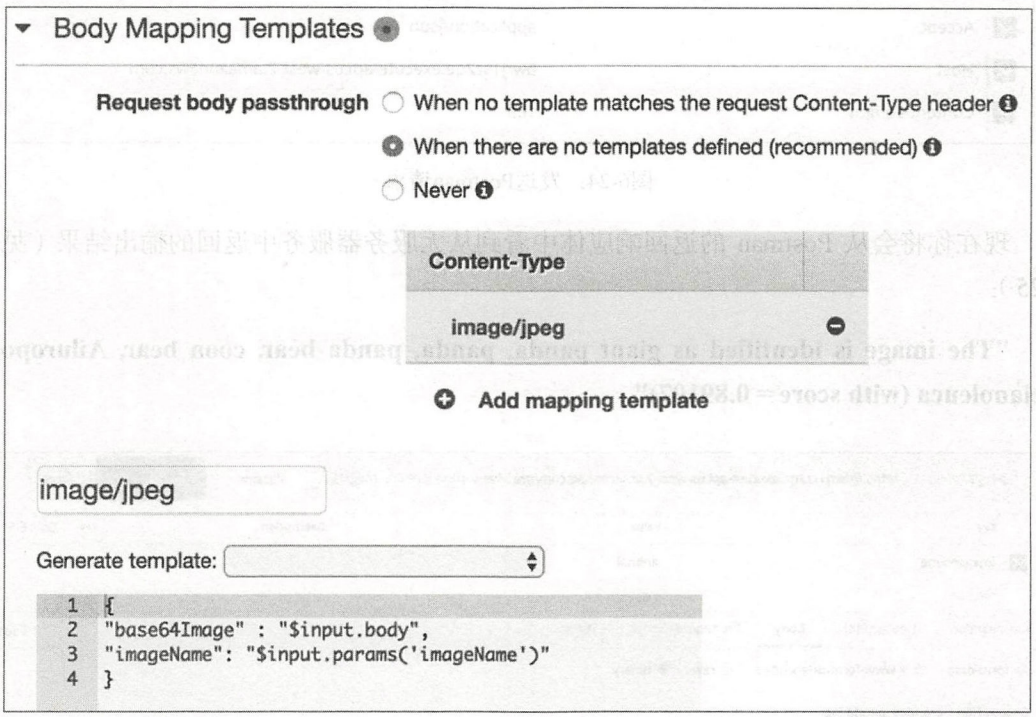


图6-23：配置API返回模板

13. 最后通过 Action 菜单定义 Stage name（例如 `prod` 或者 `dev`）来部署 API。当你的 API 部署成功后，可以通过下面的 URL 进行访问：

```
https://<API ID>.execute-api.<region>.amazonaws.com/
```

14. 接下来通过一个 REST 客户端来访问你的 API，例如通过下面例子中的 POSTMAN 来启动你的图像识别服务。在 API Request 里设置 Content-Type 为 image/jpeg，并加入参数名 imageName 及其对应的值（例如 `animal`）。在请求数据中加入图像的二进制（`binary`）文件作为我们服务要预测的内容（见图 6-24）：

```
https://<API ID>.execute-api.<region>.amazonaws.com/prod/
tensorflow-predict?imageName=animal
```

Key	Value	Description
Content-Type	image/jpeg	
Accept	application/json	
Host	bwrj1sz2qe.execute-api.us-west-2.amazonaws.com	
Content-Length	168	

图6-24：发送Postman请求

现在你将会从 Postman 的返回响应体中看到从无服务器服务中返回的输出结果（见图 6-25）：

"The image is identified as giant panda, panda, panda bear, coon bear, Ailuropoda melanoleuca (with score = 0.89107)"

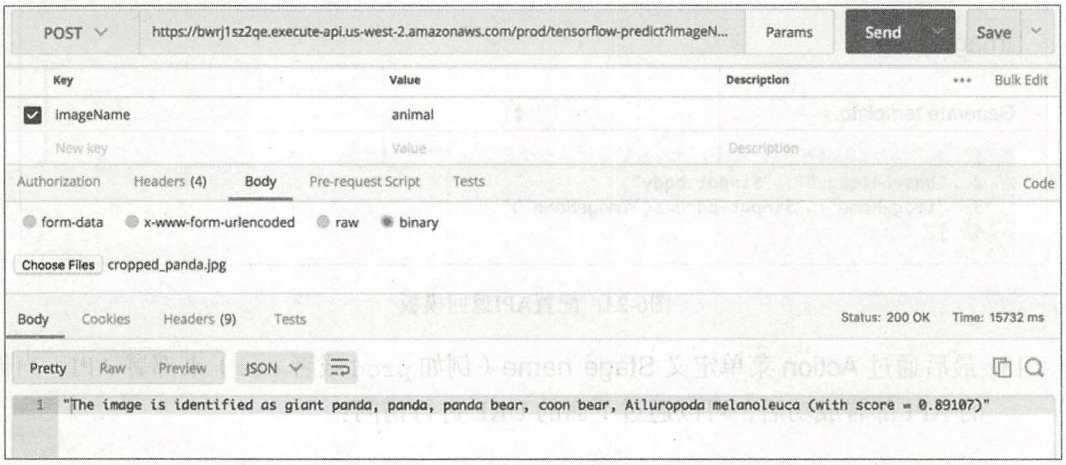


图6-25：Postman返回结果

同时在 S3 桶的 `audio` 目录下会保存生成的预测音频（见图 6-26）。

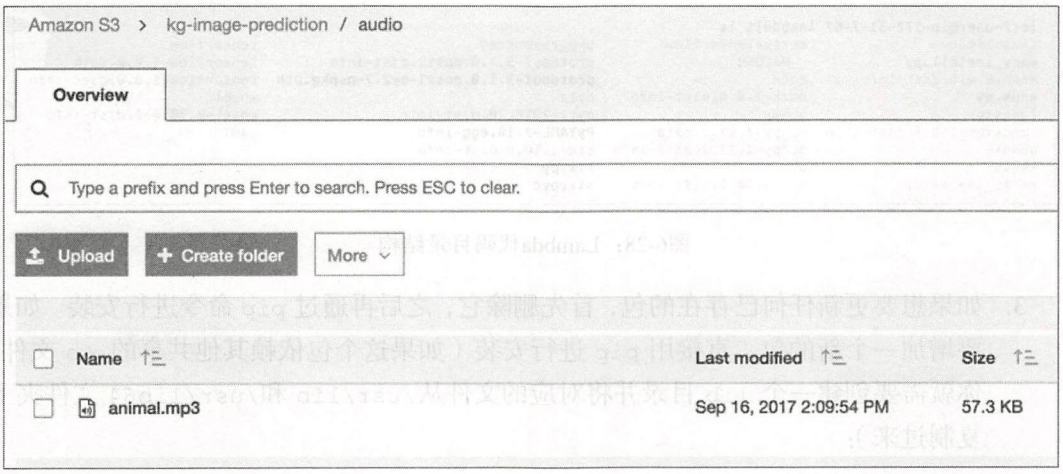


图6-26：S3中生成的语音文件

6.6.1 修改 Lambda 环境下代码和包的步骤

如果你需要增加额外的 Python 包或者更新已有的包，请执行下面的步骤。

1. 启动一个规格为 Amazon Linux AMI 2017.03.1(HVM),SSD Volume Type 的 EC2 实例（见图 6-27）。

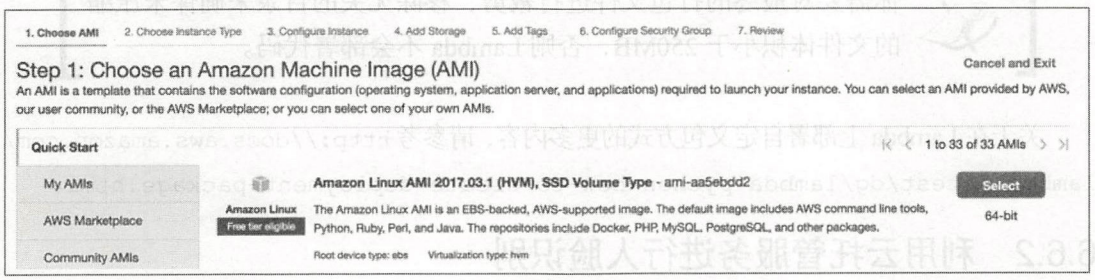


图6-27：选择AMI镜像

2. 登录 EC2 实例并将当前的 Lambda 代码复制到实例中。接下来创建一个目录并在目录内解压缩 ZIP 文件（见图 6-28）：

```
mkdir lambda
cd lambda
unzip lambda_tensorflow.zip
```

```
[ec2-user@ip-172-31-7-67 lambda]$ ls
classify.py              Keras-Tensorflow        pkg_resources                          tensorflow
easy_install.py          __MACOSX                protobuf-3.1.0.post1.dist-info         tensorflow-1.0.0.data
enum-0.4.6.dist-info     mock                    protobuf-3.1.0.post1-py2.7-nspkg.pth   tensorflow-1.0.0.dist-info
enum.py                  mock-2.0.0.dist-info    pytz                                   wheel
funcsigs                 numpy                   pytz-2016.10.dist-info                 wheel-0.30.0a0.dist-info
funcsigs-1.0.2.dist-info numpy-1.11.2.data       PyYAML-3.10.egg-info                   yaml
google                   numpy-1.11.2.dist-info  six-1.10.0.dist-info
keras                    pbr                     six.py
keras_lambda.py          pbr-1.10.0.dist-info    six.pyc
```

图6-28：Lambda代码目录结构

3. 如果想要更新任何已存在的包，首先删除它，之后再通过 pip 命令进行安装。如果要增加一个新的包，直接用 pip 进行安装（如果这个包依赖其他共享的.so 文件，你就需要创建一个 lib 目录并将对应的文件从/usr/lib 和/usr/lib64 文件夹下复制过来）：

```
rm -rf tensorflow*
pip install tensorflow==1.2.0 -t /home/ec2-user/lambda
```

4. 将整个文件夹打包成一个 ZIP 文件：

```
zip -r lambda_tensorflow.zip *
```

5. 最终将 ZIP 文件上传到 S3，并更新 Lambda 函数使用最新的 S3 文件路径。

你需要对最终的打包文件进行裁剪，移除无关的目录来确保未压缩的文件体积小于 250MB，否则 Lambda 不会部署代码。

关于在 Lambda 上部署自定义包方式的更多内容，请参考 http://docs.aws.amazon.com/lambda/latest/dg/lambda-python-how-to-create-deployment-package.html。

6.6.2 利用云托管服务进行人脸识别

在本例中我们利用基于云的机器学习服务来制作自己的标签和人脸识别系统。我们会继续使用 AWS Lambda 提供的无服务器环境并利用 AWS Rekognition 来进行面部特征识别。

执行下面的步骤来利用云上机器学习服务和无服务器平台构建人脸识别系统。

1. 更新上一个例子中的 IAM Lambda 执行角色，增加新的 AmazonRekognitionFullAccess 策略，如图 6-29 所示。

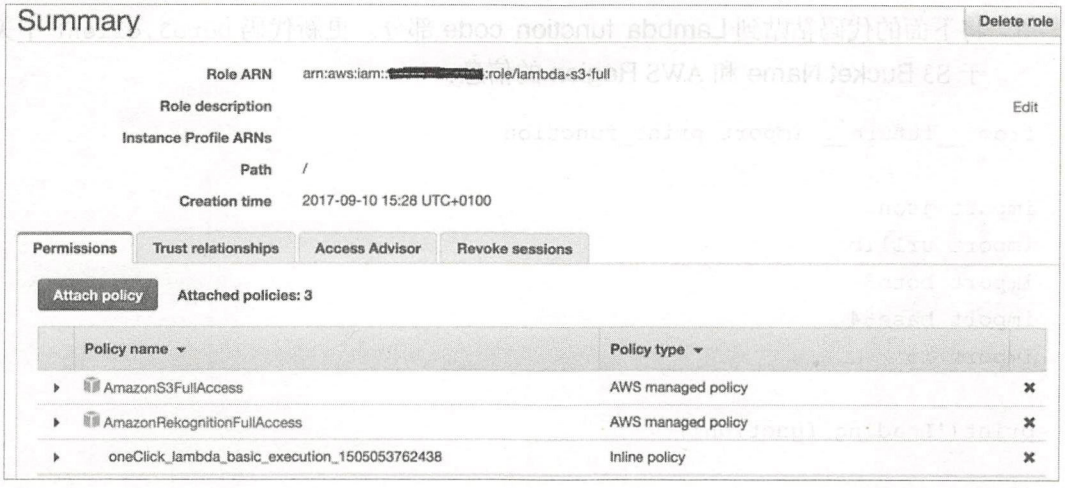


图6-29：修改IAM策略

2. 创建一个新的 Lambda 函数来构建人脸识别服务。选择 Python 2.7 作为运行环境（Runtime），其他选项保持默认值。将更新后的 IAM 角色绑定到 Lambda 函数上（见图 6-30）。

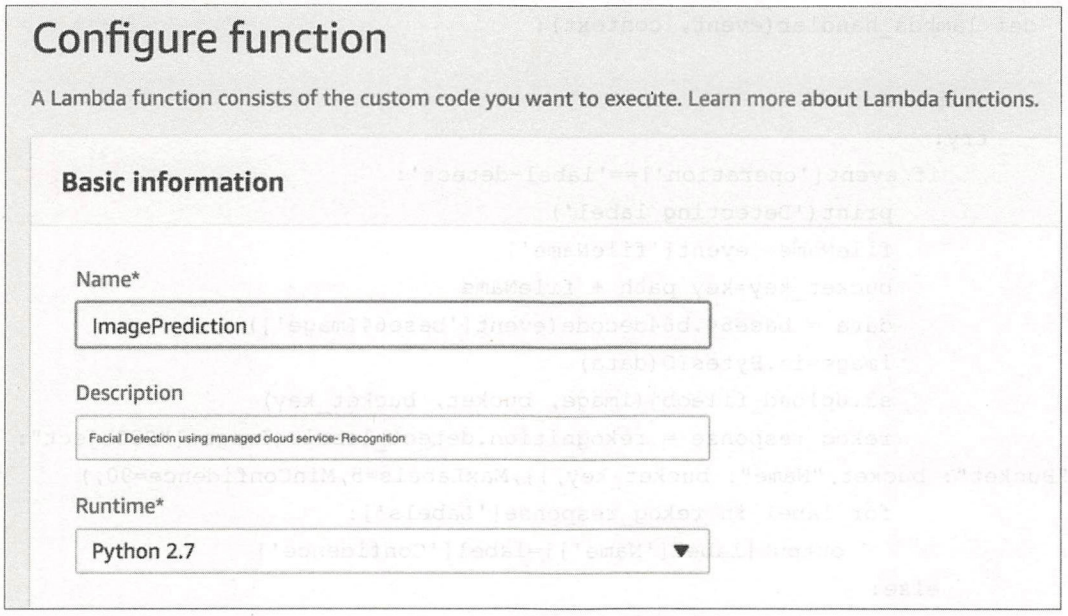


图6-30：创建Lambda

3. 将下面的代码粘贴到 Lambda function code 部分。更新代码 boto3.client 中关于 S3 Bucket Name 和 AWS Region 的信息：

```
from __future__ import print_function

import json
import urllib
import boto3
import base64
import io

print('Loading function')

s3 = boto3.client('s3')
rekognition = boto3.client("rekognition", <aws-region name like us-west-1>)

bucket=<Put your Bucket Name>
key_path='raw-image/'

def lambda_handler(event, context):

    output={}
    try:
        if event['operation']=='label-detect':
            print('Detecting label')
            fileName= event['fileName']
            bucket_key=key_path + fileName
            data = base64.b64decode(event['base64Image'])
            image=io.BytesIO(data)
            s3.upload_fileobj(image, bucket, bucket_key)
            rekog_response = rekognition.detect_labels(Image={"S3Object":
{"Bucket": bucket,"Name": bucket_key,}},MaxLabels=5,MinConfidence=90,)
            for label in rekog_response['Labels']:
                output[label['Name']]=label['Confidence']
        else:
            print('Detecting faces')
```


```
            FEATURES_BLACKLIST = ("Landmarks", "Emotions", "Pose", "Quality",
"BoundingBox", "Confidence")
            fileName= event['fileName']
            bucket_key=key_path + fileName
            data = base64.b64decode(event['base64Image'])
            image=io.BytesIO(data)
            s3.upload_fileobj(image, bucket, bucket_key)
            face_response = rekognition.detect_faces(Image={"S3Object":
{"Bucket": bucket,  "Name": bucket_key, }}, Attributes=['ALL'],)
            for face in face_response['FaceDetails']:
                output['Face']=face['Confidence']
                for emotion in face['Emotions']:
                    output[emotion['Type']]=emotion['Confidence']
                for feature, data in face.iteritems():
                    if feature not in FEATURES_BLACKLIST:
                        output[feature]=data
        except Exception as e:
            print(e)
            raise e

        return output
```

4. 创建完 Lambda 函数后创建 API 网关的子资源（见图 6-31）。

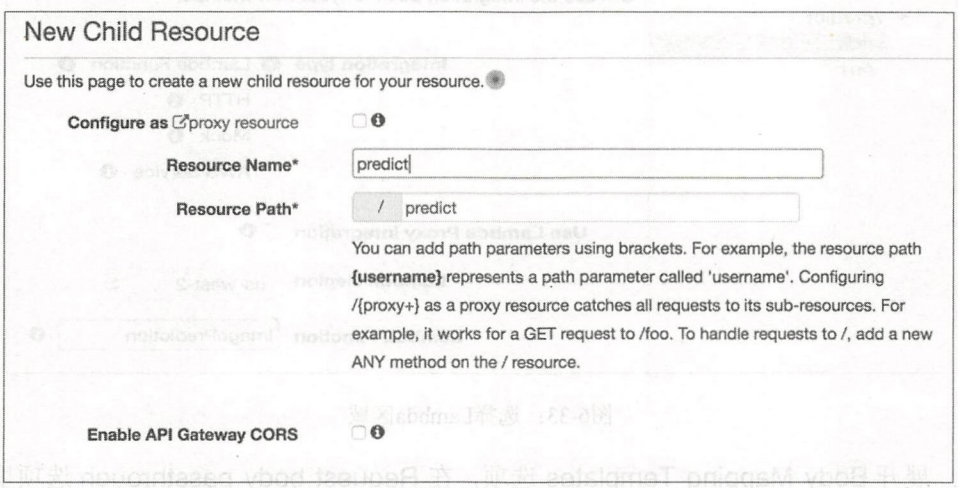


图6-31：创建API网关的子资源

5. 在新的子资源（predict）里增加一个方法（本例中为 PUT），接下来单击 PUT 方法中的 Integration Request（见图 6-32）。

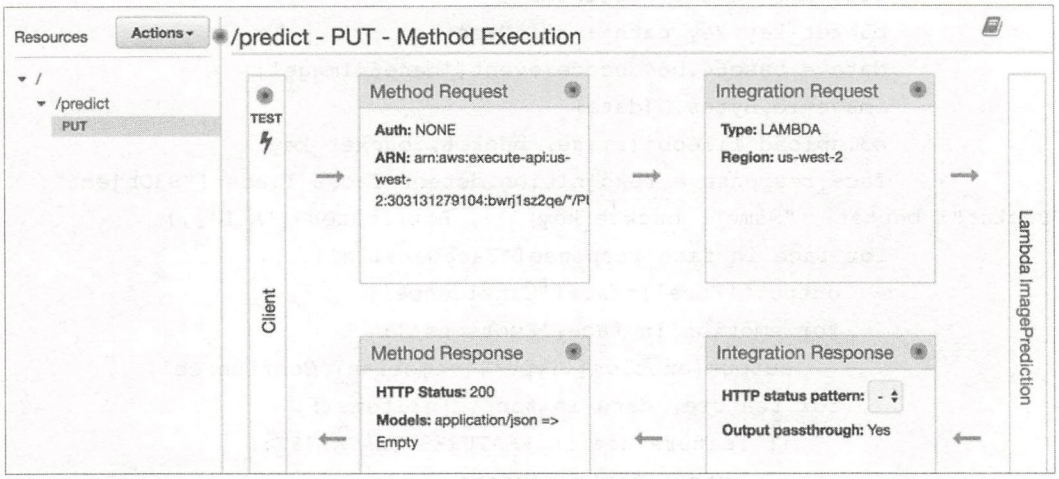


图6-32：新建PUT方法

6. 将之前创建的 Lambda Function 和资源方法绑定。你需要在 AWS Lambda Region 下拉菜单中选择 Lambda Function 所在的区域（见图 6-33）。

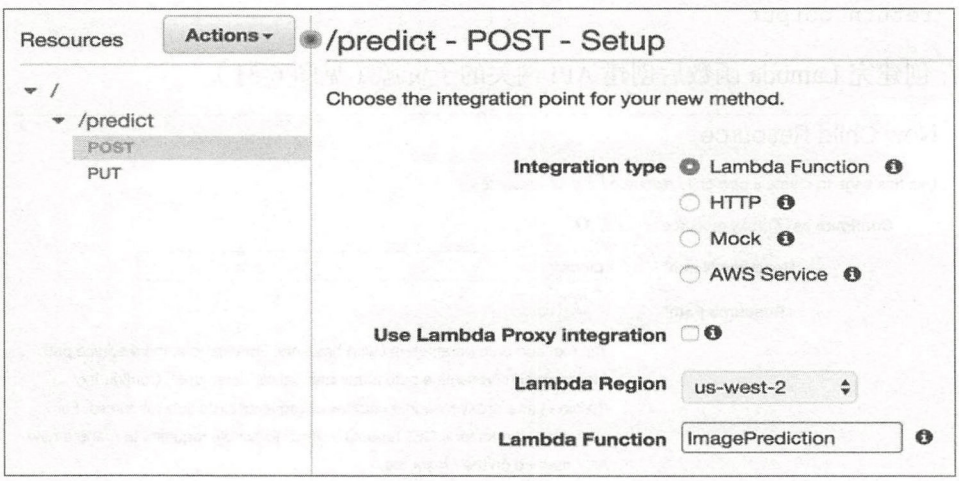


图6-33：选择Lambda区域

7. 展开 Body Mapping Templates 选项，在 Request body passthrough 选项里选择 When there are no templates defined(recommended)。接下来在 Content-Type

里增加映射模板 image/png，并将下面的代码复制到 General template 区域（见图 6-34）：

```
{
"base64Image": "$input.body",
"operation": "$input.params('activity')",
"fileName": "$input.params('fileName')"
}
```

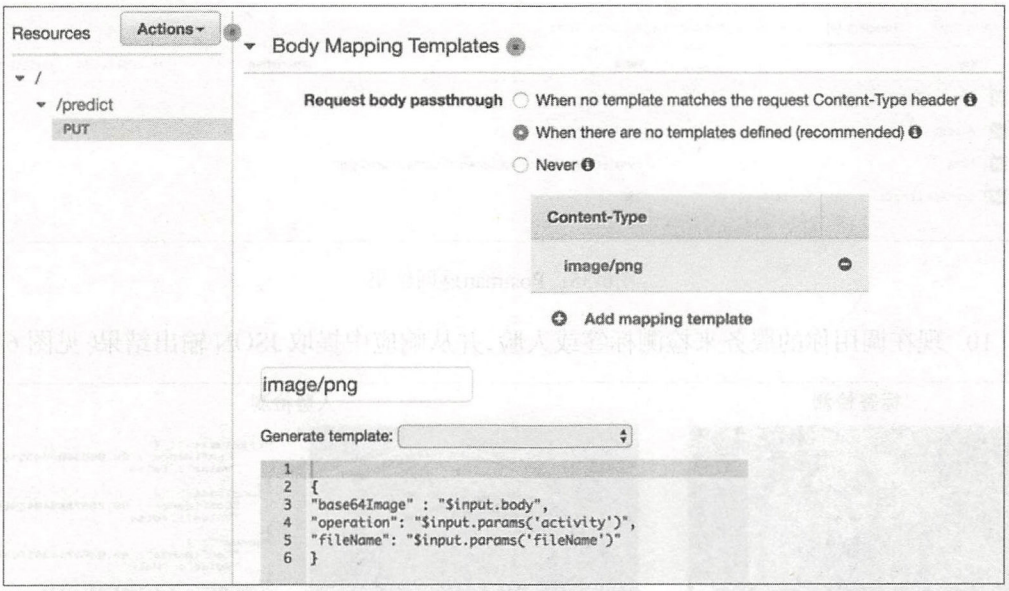


图6-34：配置API返回模板

8. 现在单击 Action 菜单下的 Deploy API 来部署你的 API Gateway 资源。当你的资源部署成功后，你将会得到一个 API 连接，可以调用人脸识别服务。我们会继续使用上个例子中用到的 REST 客户端 Postman（https://chrome.google.com/webstore/detail/postman/fhbjgbiflinjbdggehcddcbncdddomop?hl=en）来进行测试，你也可以选择任何其他的 REST 客户端。API Gateway 的连接看起来是下面这样的：

```
https://<API ID>.execute-api.<AWS Region>.amazonaws.com/prod/predict
```

9. 增加 Content-Type 为 image/png，新增两个请求参数 activity 和 filenames。参数 activity 接收两个值（label-detect 代表图像识别或者标签识别，face-detect 代表人脸识别）。参数 fileName 用来将原始图像保存到 S3 上对应的名字中（见图 6-35）。

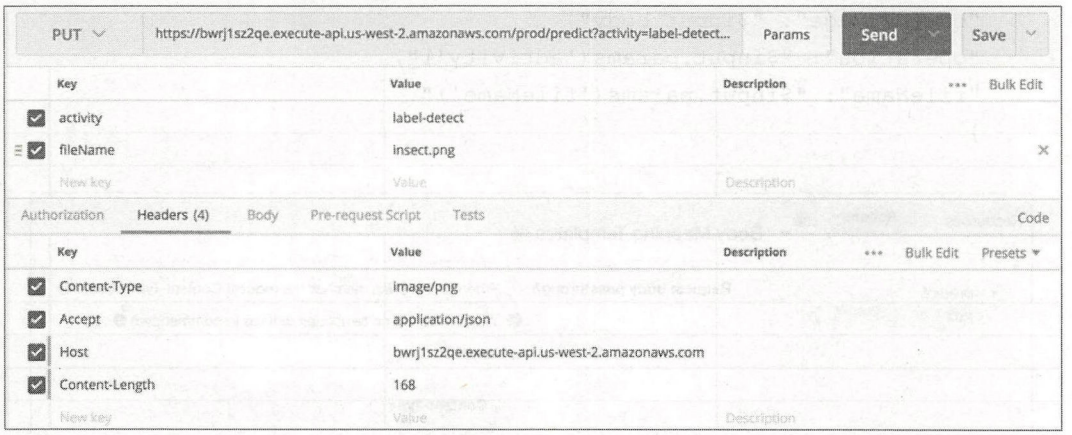


图6-35：Postman返回结果

10. 现在调用你的服务来检测标签或人脸，并从响应中提取 JSON 输出结果（见图 6-36）。

图6-36：标签及人脸检测结果示例

6.7 总结

到目前为止，你已经学习了如何通过多种方法部署训练后的深度学习模型，并根据新的样本进行预测。你也学会了如何将本地或者数据中心的模型利用 Docker 平滑地迁移到云中。希望通过学习本书中大量真实世界的公开数据集以及大量的例子，你已经理解了 GAN 的概念及其变种的架构（SSGAN、BEGAN、DCGAN、CycleGAN、StackGAN、DiscoGAN）。当你尝试使用本书中的代码和例子的时候，我十分鼓励你做下面的事情。

参加 Kaggle 对抗网络竞赛：`https://www.kaggle.com/c/nips-2017-defense-against-adversarial-attack`。

通过浏览或者参加下面的会议，使得你关于深度学习和 GAN 的知识持续保持最新：

- Neural Information Processing Systems（NIPS）：`https://nips.cc/`。
- International Conference on Learning Representations（ICLR）：`HTTP://WWW.ICLR.CC/`。

Broadview®
www.broadview.com.cn
博文视点·IT出版旗舰品牌
技术凝聚实力·专业创新出版

GAN实战生成对抗网络

使用生成模型构建的AI系统可以自动地利用源数据进行学习和理解，这使得它在数据科学家中越来越受欢迎。和监督学习方法不同，生成模型不需要对数据进行标记，这使得它成为一个十分吸引人的系统。本书将帮助你构建和分析深度学习模型并将其应用于现实世界的问题。本书还将帮助你从多个数据集（主要集中于图像领域）来开发智能和具有创造力的应用。

本书从生成模型的基础开始，带你了解生成对抗网络及其构建模块背后的理论。通过使用TensorFlow、Keras和PyTorch等库，你可以利用GAN来克服文本到图像合成的问题。在处理大型数据集时，将风格从一个域传递到另一个域会变得令人头痛。本书将使用真实世界的例子来展示如何克服这一困难。本书将帮助你理解并构建生成对抗网络模型，并在生产环境中有效、准确地使用它们。

通过本书你将会：

◎了解深度学习的基础知识，以及判别模型和生成模型之间的区别

◎利用基于生成对抗网络（GAN）和真实世界的数据集构建半监督学习模型来生成图像

◎通过利用小批量、特征匹配和边界均衡等技术来解决模式崩溃、训练不稳定等挑战，以对GAN模型进行调优

◎使用深度学习堆叠架构从文本生成图像

◎组合多个生成模型来探索跨域的关系

◎探索多种在生产环境部署深度模型的方法

上架建议：人工智能/机器学习

ISBN 978-7-121-34254-7

9 787121 342547

定价：65.00元

策划编辑：张春雨
责任编辑：李云静
封面设计：李　玲